Characterization and Design of Rational Competent Execution Agents for use in Dynamic Environments

Glen A. Reece

Ph.D.
University of Edinburgh
1994

Abstract

Carrying out previously developed plans in the real world, such as picking a friend up at the airport or cooking dinner, requires that we possess some basic abilities for behaving rationally. As we all know, things just never go to plan 100 percent of the time, so possessing the facilities to cope with this uncertainty makes us good execution agents. However, the term "rational" is easy to state, and difficult to define. Just what abilities do we possess that allow us to reason in a rational manner?

Researchers in the field of Artificial Intelligence (AI) have been attempting to make a precise definition of rationality since the beginning. Though we still do not have a universally accepted definition, we do have systems which to varying degrees appear to demonstrate qualities which can be said to be rational. If we characterize those systems, we can define a minimum set of capabilities which allow us to design a basic rational agent.

This thesis proposes a design of a reactive execution agent (REA) which accepts and executes task directives in a dynamic environment. This design draws on an amalgam of ideas emerging from situated rational agency research. It yields an agent which is able to accept task directives (or reject them), reason about directives to determine how to achieve them (maintaining commitments to achieving other directives), respond to execution failures, and communicate with a superior agent when further deliberation is required beyond the abilities of the REA.

The primary contributions of the thesis are (1) a characterized model of rational behavior, (2) an inter-agent communication language for adapting execution-time behavior, and (3) the use of causal structure to predict potential execution failures. The thesis shows how reasoning about commitment to tasks, failures related to tasks, and tasks with temporal extent interact with the basic set of capabilities to provide a robust model for competent and rational situated behavior.

Acknowledgments

First, and foremost, I would like to thank my parents for their support and encouragement throughout this degree. Living abroad has had its trying moments and they have kept me positive, full of hope, and pushed me to strive for the best future possible. I can never thank them enough.

I am also indebted to my advisor Austin Tate. Austin allowed me the freedom to pursue my research ideas, and has been a constant source of enthusiastic encouragement. I am greatful for the opportunity to have worked with him, and hope he does not think that all Americans are as crazy as I am.

I would like to thank the other members of my committee: John Hallam, Jeff H. Collins, and Brian Drabble. John brought a much needed pragmatic view to my research, and helped me get my BNF straight. Jeff advocated that I come to Edinburgh to get my degree, and spent a great deal of time and effort to make sure I got settled. Brian and I had many useful discussions about my ideas, and he kept reminding me that it was possible to get it done.

Thanks to Karl G. Kempf (Intel, AZ) for inspiring me to pursue my Ph.D., and to come to Edinburgh to do it. He said that it would broaden my views and give me a new way to look at things; it has.

A special thank you to Jim Firby (University of Chicago) for reading early drafts of this dissertation, and his support of my ideas. Jim made me believe that my research was worthwhile and could be a benefit to others.

I would like to thank many people for influencing my ideas, but especially the following people who have provided me with papers, advice, and pointers to research that I was unaware of: Mark Drummond (NASA Ames), David Wilkins (SRI), R. Peter Bonasso (Mitre), Erann Gat (JPL), Leslie Kaelbling (Brown University), Drew McDermott (Yale University), Damian Lyons (Philips Labs), and Reid Simmons (Carnegie Mellon University).

Many people have passed through room E17 during my tenure in Edinburgh. Though I cannot name them all, I thank them for the discussions, help, and/or light relief. Especially, Wamberto Vasconcelos, Ian Frank, Rob Scott, Matt Crocker and Edward Jones. Thanks also to Arthur Seaton, Jeff Dalton and Richard Kirby of the Artificial Intelligence Applications Institute for programming help and a great many laughs.

I am greatful to Abraham Waksman and the U.S. Air Force Office of Scientific Research for the grant which allowed me to conduct this research. The grant under the Augmentation Award for Scientific and Research Training program was part of the Advanced Research Projects Agency/Rome Laboratory Planning Initiative.

For having to put up with my eccentricities and neuroticisms I must thank my flatmates: Anthony Taylor, Ian Soosay, and especially Laura Tasker. I am indebted to Laura for a great many things, but most of all my sanity.

Lastly, I would like to thank my examiners: Sam Steel (University of Essex) and Louise Pryor (University of Edinburgh). I think they set a record reviewing my dissertation and having my viva approximately one month from the date I submitted.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Glen A. Reece
Edinburgh
November 24, 1994

The research discussed in Section 7.1 was previously published as:

G. A. Reece and A. Tate. Synthesizing Protection Monitors from Causal Structure. In *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pages 146-151, Chicago, IL, 1994.

Contents

List of Figures

Chapter 1

Introduction

We humans are very resilient when it comes to performing everyday activities and adjusting to failures of those activities when things do not go as we planned. A car may stall at a signal light; a taxi may arrive 10 minutes late; the local pub may run out of beer; a particular route to the airport is closed. Coping with such events requires us to modify our behavior. We may also have to carry out tasks on behalf of others which means we need to reason about our capabilities, prior commitments, and time constraints. All of these are abilities which enable us to cope with uncertainty and reason about how events effect our lives makes us rational[1].

Over the past two decades the development of general purpose planning systems which can automatically produce plans of action for subsequent execution has been the primary focus of researchers in the field of Artificial Intelligence (AI) based planning. As those techniques have begun to mature, the focus over the last few years has turned toward situated rational agency, that is, closing the loop between execution, monitoring, and failure recovery of those automatically produced plans in dynamic environments. There have been some successes with previous attempts at closing the loop, but often the plans generated were rather limited and not very flexible [Tate 89]. In general, the complexities of the individual tasks of plan representation, generation, execution monitoring, and failure recovery has led to research into each of these issues separately.

As more researchers have begun to focus on the issues involved in rational agency several themes have emerged in the AI literature. First, that rational reactive agents

[1] Potentially make us rational—there are many irrational people about.

must possess both the ability to refine previous predictions about the future based on information at run–time (i.e., deliberate), and the ability to effect changes in the environment based solely on information immediately at hand (i.e., react without further planning). Second, that the agent architecture must combine facilities from both the uniform architecture approaches where a single representation and control structure is used for both deliberation and reaction, and the layered architecture approaches where layers functionally separate representation and control structure for deliberation and reaction [Hanks & Firby 90]. Third, that neither the classical "plan then execute" nor the *reactive planning* approaches provide a completely satisfactory model of agent action [Downs & Reichgelt 92].

The hypothesis of this research is that by providing a means to build execution systems from a prescribed set of behaviors according to an established design methodology we, as a community, will be able to build quantitatively better execution systems. Therefore, what is needed is a specification of behaviors which together allow a system to behave rationally over a wide variety of complex and dynamic domains, and a methodology for integrating that specification into an architecture that addresses the emerging themes from situated rational agency research.

In this dissertation a characterization of rational behavior is presented and used as the basis for the design of a Reactive Execution Agent (REA) according to the Modeling, Analysis, and Design (MAD) methodology [Cohen 91]. At present, there are no established guidelines for developing execution systems. The characterization which results from this research is therefore intended to serve two fundamental roles: first, to provide guidelines for the development of execution systems whose behavior is to be rational in complex and dynamic environments; second, to provide a means by which execution systems can be quantitatively measured and compared against one another so we can identify which approaches are most appropriate for specific domains.

In this chapter, the objectives are defined and the approach that will be used to achieve these objectives is presented. We begin by considering examples of some agents whose behavior we would like to be able to mimic as a source of motivation. We then briefly discuss the design approach, and give an overview of the functionality of the agent we intend to design. Next we present our research methodology, and summarize the

contents of the chapters which follow.

1.1 Motivating Examples

Let's begin by considering some examples of situated systems that we would like to see developed.

I have always been of the philosophy that Laziness is the Mother of Invention. That is, technological advances are due largely to the fact that we want life to be convenient, and for activities that we must perform to require as little effort on our part as possible. This first example is adopted from an idea by [Bonasso & Slack 92] and motivated by my laziness and mediocre golf game.

Imagine that sometime in the future we will be able to order a robotic caddy that assists us on the golf course. We pull it out of the box, turn it on, and its in-built set of behaviors come online. The set of behaviors will be defined by the model that we purchased, but all models comes with the basic capabilities for locomotion control, owner following/tracking, limited sensing for obstacle avoidance, score keeping, and internal monitoring (i.e., battery level monitor, and service timer). The advantage of the new model series is that we can modify the basic caddy model to include club selection, ball trajectory tracking, swing analyzer, and course map reader capabilities. We are provided with a system whose architecture allows it to exhibit a set of behaviors enabling it to perform its duties on a golf course while being able to upgrade the basic behaviors with add-on capability modules.

The ability to adapt the behavior of a basic system to exhibit more complex behaviors is a nice idea. However, with the golf caddy we could be talking about swapping out chips, boards, or adding peripherals. Let's consider another example where we still require the abilities of the agent to be adapted, but this time there is a limitation to the level of adaptation.

Consider the design of a satellite, which we would like to have the ability to change its behavior and have it perform tasks in new ways or perform new tasks using the resources at its disposal. Imagine that our satellite is to take atmospheric samples of comets. To do this we have equipped it with the necessary sensors, effectors, thrusters,

power sources, telemetry arrays, and a telerobotic arm. The difference between the golf caddy and a satellite it that once the rocket lifts off from the launch pad, we are limited in how we can adapt its behavior because we can no longer swap out chips and boards. We can sit here on earth and think about many situations that the satellite might find itself, but we surely cannot think of every possibility. Say that, once the satellite had been launched, we determined a way to increase the effectiveness of a sensor, learned how to conserve energy, or how to use some of the existing capabilities together to get a new capability. What we would need would be a way to communicate these changes so they could be used by the satellite.

Let's consider one more motivating example, but this time the system in question is charged with dispatching tasks to resources under its control which themselves have particular capabilities. For this example, we will discuss a robotic cell controller whose job it is to accept a work plan for the assembly of a part. The controller is charged with controlling a conveyer-belt, a five degree of freedom robot, a three degree of freedom robot, and an autonomous effector delivery robot.

When an assembly plan is given to the cell controller it determines which resources are required, schedules those resources, and dispatches the appropriate subtasks of the plan to the resources. This is akin to a command and control scenario where tasks are decomposed and passed through layers of control. As tasks fail or complete, communication events between the resources dictate what is to be done next.

1.2 Approach

The research presented in this dissertation is mainly an exploratory investigation into whether we can characterize rational behavior and design an agent that competently exhibits that behavior in complex and dynamic environments. To achieve this goal, we take a top-down approach to the design.

We begin by characterizing existing execution systems to establish a basic set of characteristics that together provide the competency to behave rationally. Then we use an agenda–based architecture as the means to integrate these characteristics, and test this approach in a realistic simulated environment. This shows that the characteristics

and be integrated together to give the desired behavior, and validates the fact that an agenda–based architecture is an appropriate vehicle for developing such systems. We then re–evaluate the characterization based upon what we have discovered during this process so as to establish a guideline for the design of subsequent reactive execution systems.

In this section we briefly introduce the characterization, the design of the architecture, and the evaluation testbed which will be further elaborated in the remaining chapters of this dissertation.

1.2.1 The Characterization

In order to design a system that performs rationally in complex and dynamic environments we need to quantify rationality. We need to be able to identify those characteristics of behavior that together allow a system to exhibit rational behavior. We can then specify that any system that is to behave rationally while situated in such environments must at a minimum possess these characteristics.

However, such a characterization is not static. That is, we can specify a set of characteristics that together characterize rational behavior, but that characterization needs to evolve as more requirements are made on these agents. Therefore, the characterization forms the basis of a design cycle. We present in this research one possible characterization of rational behavior. We then evaluate that characterization against a representative group of existing systems to validate it as representing characteristics of rational behavior. From this characterization we then design an architecture that will allow these characteristics to be integrated together and evaluate that architecture in a realistically complex and dynamic environment. Next, we re–evaluate that characterization to determine if the definitions of the characteristics are unambiguous and explicit so that the degree to which one system possess a specific characteristic can be better determined. With this improved characterization the cycle should begin again with the design of future systems using it as their base requirements.

1.2.2 The Design

We envision the REA being a self-sustaining entity that is required to carry out tasks in the environment in which it is situated. It possesses the ability to execute tasks concurrently and utilizes causal information to monitor the execution of tasks it is to achieve. It is able to recover from anticipated failures and to request assistance in the form of additional knowledge, capabilities, and tasks from a superior agent when it finds itself in situations where its knowledge and capabilities are inapplicable. The REA also responds to changes in the environment and performs other tasks without an explicit plan. That is, it possesses innate behaviors that help it to survive in its environment.

The design of the architecture for the REA is dictated by the requirements of the characterization. In addition to integrating the mechanisms necessary to implement each of the characteristics, we wish to explore the issues of inter–agent communication, adaptability, and execution monitoring. These latter issues place additional requirements on the design.

To meet the requirements and make the vision of the REA come to fruition, we utilize an agenda-based architecture that encapsulates domain and processing knowledge in the form of knowledge sources. In this architecture, two types of events occur—task and environmental. When an event is detected, it is placed on the REA's list of active intentions to be processed by a capability appropriate for the type of event. Task events are processed according to (1) the priority of the capability that must be used to process them and (2) the commitment level for the task. Environmental events are processed according to the priority of the capability necessary to process them.

1.2.3 The Testbed

In order to evaluate an agent architecture with capabilities such as concurrent task execution, remote sensing, and the ability to execute tasks over extended durations we require a simulator that has features beyond any one simulator which is known to exist. We also require a domain that is realistically complex and dynamic enough to fully exercise all aspects of the design.

To meet these needs we have designed the Pacifica Simulator that simulates a transportation logistics domain. The simulator provides a complex environment by simulating weather, terrorist activity, and natural disasters that take place on the fictitious island of Pacifica.

1.3 The Reactive Execution Agent

In seven of the eight chapters which follow, we will examine in detail how behavior of the REA is defined and controlled. However, we will first consider an overview of that behavior here in order to have a foundation for those later discussions.

The focus of this research is in dynamic domains where the demand is for a system that can take a command request, generate a plan, execute it and react to simple failures of that plan, either by repairing it or by re-planning [Tate *et al.* 92]. This is investigated in the context of two agents with different roles and with possible differences of requirements for processing capacity (Figure 1.1[2]). Here, the two agents involved will be a planning agent and a remote Reactive Execution Agent (REA).

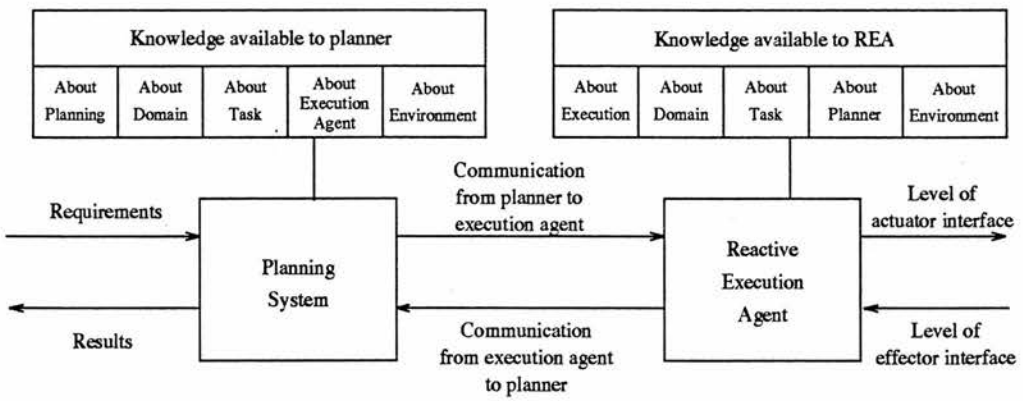


Figure 1.1: Communication between Planner and Reactive Execution Agent

The REA is a separate process that possesses domain knowledge of the environment in which it is situated and capabilities that allow it to execute tasks, control resources, and communicate with the environment and a planning agent. The planning agent is responsible for generating plans that the REA is to carry out in the environment and

[2] This figure is a modified version of that presented in [Tate 89].

to assist the REA when circumstances are beyond its knowledge and capabilities to address.

The REA seeks to carry out the detailed tasks specified by the planning agent while working with a more detailed model of the execution environment than is available to the planning agent. It executes the plan by choosing the appropriate activities to achieve the various sub-tasks within the plan, using its knowledge about the particular resources under its control. It communicates with the real world by executing the activities within the plan and responding to failures fed back from the environment. Such failures may be due to the inappropriateness of a particular activity, or because the desired effect of an activity was not achieved due to an unforeseen event.

In this research, a new control strategy is not proposed or developed. There are enough good robotics researchers developing these strategies. The purpose here is to design a representation and control mechanism based upon an approach which has proven successful (i.e., Firby's Reactive Action Packages [Firby 89]), and implement it in a new type of architecture (i.e., the agenda–based architecture of O–Plan [Tate *et al.* 94]) that allows for adaptability, modularity, and add some additional features not present in the original approach. Hopefully, these new features will lead to more robust, intelligent control that will be adopted in future control strategies.

Communication between the planning agent and the REA are in the form of messages defined in the Inter–Agent Communication Language (Chapter 5). The IACL *synthesize* message is the means by which the planning agent provides information to the REA and directs it to carry out a plan. This message contains high–level descriptions of the tasks which together compose the plan, ordering constraints on how the tasks should be executed, commitment information instructing the REA as to how crucial the execution of the plan is, and causal structure that allows the REA to reason about the causal relations between tasks in the plan. When the REA receives an IACL *synthesize* message it uses the information in the message to synthesize a Task Directive which it can then reason about and execute.

The domain knowledge of the REA is represented at a lower level of abstraction than that which the planning agent employed when generating the plan. There is no need for the planning agent to reason about all of the subtasks and sensing necessary to fly

a cargo plane from one location to another. All it should need to concern itself with is that there are resources at one location which need to be moved to another and that an REA is available that possesses the knowledge to achieve such a task. Therefore, the tasks specified by the planning agent are decomposed by the REA according to behavior specifications of the Task Behavior Language (Chapter 3) into more detailed tasks which can be carried out in the environment.

The intentions of the REA are conceptually maintained in two prioritized lists called the triggered agenda and the untriggered agenda. Those intentions on the triggered agenda are currently activated and being considered for processing by the REA's capabilities. The intentions on the untriggered agenda are ones which are suspended (i.e., not activated) and are awaiting some condition to be satisfied according to the REA's model of the environment.

When an intention is formed it contains information specifying its priority, the capability required to process it, any necessary data, and if appropriate, control information that assists the controller in identifying particular intentions. Intentions are formed as a result of an external event, by the processing of other intentions, or as a result of an internal event.

Tasks to be executed by the REA are either primitive or a network of subtasks whose successful completion satisfies the parent task. If the task is a primitive, which means that it is specified to a level of abstraction that can be carried out in the environment, then it is dispatched to the hardware[3]. If the task is a network, then each subtask which is eligible to be executed according to temporal and ordering constraints is posted to the triggered agenda as an active intention. The remaining subtasks are posted to the untriggered agenda as a network which will become activated when the previously dispatched subtasks have successfully completed.

The REA assimilates information from task completion reports, task failure reports, and sensor reports into its internal World Model. All decisions which are made by the REA concerning its environment are made by querying this World Model.

[3] For the research presented in this dissertation, the dispatched task is sent to the Pacifica Simulator (Chapter 6) and not directly to hardware though the interface would be the same.

In addition to controlling the execution of tasks, the REA architecture possesses mechanisms to monitor potential protection interval violations, actively and passively initiate behaviors, and to actively update information contained in its World Model related to causal relations between tasks.

1.4 Research Methodology

As Paul Cohen has argued [Cohen 91], the bulk of AI research suffers from methodological problems: lack of evaluation, lack of hypothesis and predictions, irrelevant models, and weak analytic tools. To this end, he proposed MAD: a methodology of modeling, analysis, and design. The seven activities of MAD meant to alleviate methodological problems are: (1) assessing environmental factors that affect behavior; (2) modeling the causal relationships between a system's design, its environment, and its behavior; (3) designing or redesigning a system (or part of a system); (4) predicting how the system will behave; (5) running experiments to test the predictions; (6) explaining unexpected results and modifying models and system design; and (7) generalizing the models to classes of systems, environments, and behaviors.

This section begins the description of how the MAD methodology is used in the design of the REA. Throughout this dissertation the seven points of the MAD methodology will be addressed (in Chapters 1, 2, 4, 6, and 8). These sections will be at the end of the relevant chapter and identified by a title of "MAD – <activity>," where activity is one of the seven points of the methodology. We begin here with a discussion of the assessment of environmental factors that affect the behavior of the REA.

1.5 MAD – Environmental Assessment

In order to exercise the proposed design and demonstrate how the agent's capabilities interact to produce rational behavior, the simulated dynamic domain of Pacifica is utilized.

Pacifica models a command and control environment in which the agent takes the role of a commander receiving mission directives from a superior agent and issues orders

to entities under its direct control to carry out those missions. These missions are in the context of Non-combatant Evacuation Operations (NEOs). Such operations are undertaken to provide rapid response to a variety of circumstances such as storms and other natural disasters, evacuation of civilians from trouble zones, policing and medical missions, humanitarian relief, etc. They are often characterized by the need for rapid deployment of equipment and personnel to ensure that effective aid is offered and to seek to minimize the escalation of the problems through delays. The transportation logistics associated with NEOs present many interesting problems for reactive execution. The primary reason for the use of Pacifica is to demonstrate the early failure detection, resource reasoning, commitment reasoning, reflexive knowledge, communication, change-of-focus, asynchronous input, and failure management capabilities of the REA. Pacifica models events which may take several minutes, hours, or even days to complete and which may interact. It allows for concurrency, uncertainty, and exogenous change. Concurrent actions are modeled in Pacifica to demonstrate the behavior of the REA when addressing multiple, possibly interacting, task directives over extended periods of time where feedback of failure/success may not be immediate.

1.6 Outline of Chapters

Chapter 2 - Characterization of Rational Behavior

Over the last decade many researchers have begun to focus on the issues involved in rational agency. The result has been a wide variety of approaches to designing reactive and rational agents, and though each approach was usually designed to address a specific research issues, there is a great deal of common ground. Thus, there are important questions which need to be addressed. First, given all the possible choices of features an agent may possess, can we define a minimum set of characteristics which are desirable to achieve rational behavior? Second, can we design a reactive execution system which incorporates the best features from both the classical predictive and reactive planning approaches?

This chapter characterizes nine major systems from the literature relating to situated rational agency to identify a set of basic requirements for the design of rational agents.

Each of the systems is then rated according to the characteristics they possess and the results of this "comparison" are given.

Chapter 3 - World Model and Task Behavior Language

Chapter 3 discusses the organization of the REA's database which is used to model its environment, and the language used by the REA for specifying the behavior of tasks at execution time. It describes the functionality of the World Model, how sensor information is assimilated into the model, and the interface to retrieve information stored in the model. It then describes the syntax and semantics of the REA's language for specifying task execution behavior.

Chapter 4 - REA Architecture and Control

In Chapter 4, the control and processing cycles of the REA are described. It ties together the concepts introduced in the previous chapters by describing how each of the capabilities interact to allow the agent to behave rationally. Specifically this chapter discusses how the agenda mechanism, triggering, world model reasoning, and agent capabilities work to execute tasks. This chapter also shows how the REA reasons about priority of tasks, commitment to tasks, and its knowledge and capabilities.

Chapter 5 - Inter Agent Communication

Chapter 5 introduces the Inter-Agent Communication Language (IACL). IACL is a simple message protocol that allows a planning agent to communicate with the REA and specify tasks which the REA is to carry out on its behalf. IACL is not a formal specification, but a simple means to provide information and functionality to the REA that is not directly provided by other languages/protocols such as ACT [Wilkins & Myers 94], TF [Art94] or KRSL [ISX93]. The various message types and their purposes are described.

Chapter 6 - Experimentation Testbed and Evaluation Criteria

The design is tested using the domain of Non-combatant Evacuation Operations (NEOs). The transportation logistics associated with NEOs present many interesting problems for reactive execution. Chapter 6 will introduce NEOs, describe the specific scenario that will be used to demonstrate the various abilities of the REA design, and describe

the Pacifica Simulator.

Chapter 7 - Failure Management

During the execution of a task directive, failures can occur because required resources are not available, an event does not have the desired effect(s), or changes in the environment have occurred causing preconditions not to be satisfied. Chapter 7 describes the mechanisms in the REA for managing these types of failures. It shows how the REA uses monitors, as well as, active and passive behaviors, to detect and address failures. It describes how causal structure in plans from a planning system can aid the REA in detecting potential failures early to give the planning system more time to instigate a repair.

Chapter 8 - Execution Examples

Chapter 8 demonstrates how the REA functions in a complex and dynamic environment. A Non-combatant Evacuation Operation Scenario is presented which exercises various features and capabilities of the REA design. Once the individual characteristics have been demonstrated, we demonstrate other features of the design which include active sensing, active and passive behaviors, inter–agent communication, and causal structure based interval protection monitors.

Chapter 9 - Conclusions and Future Research

Finally, Chapter 9 states the conclusions which have been drawn as a result of exercising the REA design based on the characterization of rational behavior. We identify the areas of future research in regards to the designed system and to the field as a whole. Then we reconsider the definitions of the characterization to make sure that they explicitly define the desired behaviors. Finally, we extend the characterization to include characteristics that have been identified as important as a result of this research.

1.7 Chapter Summary

The topic of this research is primarily concerned with how to design a reactive execution agent. However, contributions of the work vary from what features the design of such an agent should possess to specific implementation ideas on how to improve the process

of execution monitoring. This dissertation contributes to the development of reactive execution agents in the following sense:

1. Proposes an explicit characterization of rational behavior. This characterization is a specification of the type of behavior that an agent architecture should provide if it is to yield an agent that behaves rationally in complex and dynamic environments over a wide variety of domains;

2. Proposes to validate an agenda–based architecture that utilizes knowledge sources for control and processing decisions as a flexible means to integrate characteristics of rationality that is dynamically adaptable and modular;

3. Proposes a communication language that provides a means to task an execution system, provide new or modified domain knowledge, and dynamically adapt the processing knowledge and capabilities of the system at run–time;

4. Proposes a method of synthesizing protection monitors from causal structure information and a means to allow these monitors to be used to identify potential execution failures. This latter concept allows the execution system to detect potential failures early so as to provide a planning system (which must address the failure) with a greater amount of time to initiate a repair plan.

Chapter 2

Characterization of Rational Behavior

The primary issues in designing an agent that is to behave rationally in dynamic environments are choosing architectural features (e.g., interrupt handling, reactivity, etc.) required by the domain and determining how the agent's beliefs, goals, and intentions will effect its deliberation in deciding what to do next. Thus, there are important questions which need to be addressed: questions such as, given all the possible choices of features an agent may possess, can we define a minimum set of characteristics which are desirable to achieve rational behavior?

As the environments become more complex, with possible risk to human life, agents which operate in those environments must become more rational and competent. They must act appropriately depending upon the situation in which problems arise and comprehend the future contingencies arising from those actions. Thus, another question which must be addressed is how to design rational agents which are capable of deliberation and real-time reactivity while situated in dynamic environments.

In order to answer these questions we begin in this chapter by considering the characteristics of rationality in execution systems that have been developed over the past twelve years. The objective is to define a basic set of features which together allow a system to demonstrate behavior that is rational.

With this set defined, we compare a representative sample from the literature of the most significant (based on citations) systems from integrated approaches to the recent

three-layer architectures coming out of the mobile robot community.

This chapter begins by defining eight characteristics that together provide a minimum taxonomy of rationality for reactive execution systems. It characterizes previous designs of reactive systems both from the uniform and layered architectural approaches [Hanks & Firby 90], and this characterization forms the motivation for the design of the REA (Chapter 4). The first section defines the basic characteristics of rational behavior as identified in the literature. The next section discusses related work from the AI literature and evaluates how those systems exhibit the defined characteristics and details the degree to which each characteristic is satisfied. This is followed by a tabular comparison of the results that is based upon a "three-star" rating system.

2.1 Rational Agency Characterization

In order for agents to act rationally and predictably in complex and dynamic environments they must possess a set of basic architectural and algorithmic capabilities. Capabilities such as those defined by [Laffey *et al.* 88] for real–time knowledge–based systems provide a good initial set. That set includes requirements (or capabilities) for guaranteed response times, handling asynchronous events, weighing alternatives to manage uncertainty, a change of focus-of-attention mechanism, predictability, continuous operation, and a temporal reasoning facility. However, the continuous operation capability appears to be ambiguous as to whether that capability is present when the system is able to recover from failures and continue to operate, or when its able to continue to operate due to built–in innate behaviors and act without deliberation. To remove that ambiguity, the failure recovery and innate behavior capabilities will be separated. This basic set of capabilities is defined as follows[1].

Guaranteed Response The ability to guarantee some kind of response by the time the response is required. This is typically referred to in the literature as bounded response. However, here we will relax the definition and allow for the ability of the agent to provide default or random responses for short deadlines. The response

[1] The definitions given in this paper are those of the author, and not necessarily those given in [Laffey *et al.* 88].

need not necessarily be the "best" response, but the best response given the deadline (*cf.* anytime algorithms [Boddy & Dean 89]).

Failure Recovery The agent is able to continue to function after a failure has been discovered due to its action or due to an exogenous event.

Innate Behavior The ability to act without a plan. That is, the agent possesses the innate abilities required to survive in the environment without requiring a plan. This is necessary since situated agents may not always have a plan of action to execute, yet they must continue to operate until such time a plan is given to them to carry out in the environment.

Asynchronous Events The agent must support asynchronous inputs. As the environment in which REAs are to be situated has the potential for rapid change, the agent must possess the ability to address these changes as they occur. That is, the agent must have the ability to accept new tasks while addressing others just as humans do. For example, consider driving to work. You are able to drive your car down a busy street while acquiring new information from the radio as to road conditions further along your route. You don't stop driving in order to process this information, and yet are able to use it to base future decisions upon. The REA must have this same ability.

Weighing Alternatives The ability to select alternative courses of action in the face of uncertain or missing information and varying importance of events.

Change of Focus The agent is able to focus processing attention on important tasks even if the processing of less important tasks must be rescheduled or aborted. Since the environments in which REAs are situated are dynamic, more immediate tasks may appear that require the agent's attention. For example, if the agent is a spacecraft and currently taking pictures of Mars, the fact that an asteroid is about to collide with the agent should be of more immediate concern than taking pictures. Thus, that agent must have the ability to stop processing its current task and address the more immediate task. Then, once the immediate task has been addressed, the agent should attempt to resume the prior task if the environment is in such a state that it can do so.

Predictability The ability to provide deterministic response even though the course of producing that response is non-deterministic (i.e., same circumstances implies same response). That is, for a given time constraint, the ability of the system to provide a response is determinable for a given situation. This way other agents which must intact with the REA can reason about its behavior in various situations.

Temporal Reasoning The agent is able to reason about time. That is, the agent is able to understand how to schedule actions such that they take place after events, before events, and during event intervals according to time constraints.

2.2 Representative Execution Systems

Over the past several years a substantial body of work has emerged relating to situated rational agency. That work is composed of system approaches and approaches which address specific issues relevant to this area. Here the focus is on the system approaches in order to characterize them and determine if the particular features of Section 2.1 define rational behavior. The system approaches are primarily represented by the systems developed by Richard Fikes [Fikes *et al.* 72a], Jim Firby [Firby 87], David Wilkins [Wilkins 85b], Michael Georgeff & Amy Lansky [Georgeff & Lansky 87b], Leslie Pack Kaelbling [Kaelbling 88], James Sanborn & James Hendler [Sanborn & Hendler 88], Reid Simmons [Simmons 90], John Bresina & Mark Drummond [Bresina & Drummond 90], Damian Lyons & A. Hendriks [Lyons & Hendriks 92], and Erann Gat [Gat 91].

These systems will be presented in chronological order according to their first appearance in the literature. A short description is given for each system along with an examination of the REA characteristics (from those described in Section 2.1) the particular system possesses or has the built–in ability to provide.

2.2.1 PLANEX

The STRIPS planning system, through the use of the PLANEX system, was the first attempt to address the issues involved in executing and monitoring a previously pro-

duced plan of action [Fikes *et al.* 72b]. The primary contribution from this work was a tabular plan representation structure called the *triangle table* [Fikes 71]. The *triangle table* representation was developed to assist the PLANEX system in answering questions such as *(i)* is execution proceeding as planned, *(ii)* what needs to be executed next in order to accomplish the given task, and *(iii)* can execution take place in the current environment.

A *triangle table* is a lower triangular array where rows and columns correspond to the operators of a plan [Fikes *et al.* 72a]. PLANEX extracts a "kernel model" for each action in the plan produced by STRIPS which contains only those statements STRIPS anticipated during plan generation would be true when the action was to be executed in the environment and inserts these into the triangle table. The columns of the table are labeled with the operator names from the plan and the cells contain the operator add–list information for each subsequent action which was not deleted by the application of the previous operator.

During execution PLANEX finds a kernel model whose statements can be proven in the current environment starting at the final kernel (i.e., the one which accomplishes the task) and proceeding toward the initial kernel. In other words, the execution strategy is to check the current environment to see if the task to be executed has been accomplished. If it has then execution has been successful, albeit by no action of its own, and execution halts. If it has not been accomplished then the plan is searched backwards to see if the effects of an action in the plan are currently true in the environment. If that is the case then the next action in the plan whose effects satisfy its execution preconditions is executed. This searching process continues until an action is found which can be executed or until the initial action to be executed in the plan is reached.

The PLANEX execution system possesses the following REA characteristics.

Guaranteed Response PLANEX can only produce a response when it has a plan from STRIPS and has converted that plan into kernel models and the *triangle table*. Therefore, since it could not produce a response without a plan it is not possible for it to guarantee a response.

Failure Recovery If execution fails to proceed as intended then either re–execution of some portion of the plan takes place or control is returned to the planning system to re–plan actions to accomplish the task. Though simplistic, PLANEX does perform failure recovery.

Innate Behavior The system does not provide this capability. Only when kernel models are available can PLANEX perform any action.

Asynchronous Events PLANEX does not allow for asynchronous events.

Weighing Alternatives PLANEX does not weigh alternatives. It is simply given a plan to execute and monitor from STRIPS.

Change of Focus Since PLANEX is unable to accept asynchronous events and only has knowledge of the plan it is currently working on it is unable to change its focus of attention.

Predictability According to [Fikes 71], if the axioms in a kernel model K_i are true in the current environment and if the effects of action executions are as indicated in the action descriptions, then the sequence of plan steps $i, i+1, \cdots, n$ is executable and its execution will complete the task. Hence, PLANEX is predictable for a given plan.

Temporal Reasoning PLANEX has no temporal reasoning capabilities.

2.2.2 Procedural Reasoning System

The System for Interactive Planning and Execution Monitoring (SIPE) developed by David Wilkins is a domain-independent, heuristic system which plans then monitors the execution of a plan [Wilkins 85a]. Recently, SIPE was integrated with the Procedural Reasoning System (PRS) through the ACT Formalism [Wilkins 93a,Wilkins 93b] to combine the complementary capabilities of both systems. This section will primarily address PRS for purposes of the reactive execution agent feature comparison.

PRS is a reactive planning and reasoning system designed by Michael Georgeff, Amy Lansky, and Felix Ingrand [Georgeff & Lansky 87a,Georgeff & Lansky 87b,Georgeff & Ingrand 89]. It uses a partial planning strategy, procedural knowledge of a particular

domain (in the form of Knowledge Agents), and has meta–level reasoning capabilities all in a reactive framework which guarantees response within bounded time intervals. In PRS, Knowledge Areas (KA) are declarative procedure specifications (represented as a graphic network) which tell the system how to accomplish goals and react to certain situations. This approach is similar to operators or plan schema seen in earlier planners such as NOAH [Sacerdoti 77], Nonlin [Tate 77], and SIPE [Wilkins 85b]. However, KAs are implemented more in the fashion of knowledge sources found in blackboard or agenda-based systems, with triggering and contextual components as opposed to simple pattern matching schemes seen in earlier planners. Some KAs in the PRS system include information about the manipulation of the beliefs, desires, likelihood of success, and average execution time. Such KAs are known as meta–KAs which select between KAs when several are available to accomplish a particular task, to assign priorities to tasks, or even to post new tasks to be accomplished. The intention structure approach of PRS gives the system the ability to notice newly posted events after every primitive action taken which gives PRS a guaranteed reactivity level of 5 events per second [Georgeff 89].

The PRS system possesses the following REA characteristics.

Guaranteed Response The event-based fashion with which the inference mechanism in PRS has been designed allows the system to have a guaranteed response delay. Response, in this case, is the selection of an applicable KA to address an event.

Failure Recovery When a failure occurs, the system reestablishes the goal and tries again. This continues until either the goal is satisfied or it determines the goal cannot be readily accomplished. PRS does not possess the ability to try different methods to achieve the goal other than to simply retry its execution.

Innate Behavior PRS does not provide for innate behaviors nor is it clear how such a mechanism could be incorporated.

Asynchronous Events The intrinsic goal type of PRS allows "outside" sources to impose arbitrary goals on the system through its database. Also, through the use of meta-KAs and the system interpreter PRS allows KAs on the intention structure to become active upon changes in the beliefs held by the agent.

Weighing Alternatives Though the means by which this is implemented in PRS is not explicitly discussed in the available papers, it is noted that meta–KAs choose between KAs when several are available and relevant.

Change of Focus Not all options considered by PRS arise as a result of means-end reasoning. Changes in the environment may lead to changes in the system's beliefs, which in turn may result in the consideration of new plans that are not means to any already intended end.

Predictability The PRS does follow non-deterministic paths through the KAs to arrive at deterministic results for various events and beliefs, yet it generates predictable responses in similar situations.

Temporal Reasoning Goals in PRS are expressed as conditions over some sequence of world states. PRS does not allow for explicit temporal windows of action execution to be specified but it does use temporal relationships.

2.2.3 Goals As Parallel Program Specifications

The Goal As Parallel Program Specifications (GAPPS) system developed by Leslie Kaelbling, is a compiler which translates goal reduction rules into directly executable circuits [Kaelbling 88]. These circuits, which map inputs and current situation information into an output (i.e., an action), are synthesized from high-level expressions in the REX language [Rosenschein & Kaelbling 86]. Basically, the GAPPS approach is an updated version of Fikes' *triangle table* idea with the additional features of guaranteed reaction times, multiple goal types, conjunctive goal handling, prioritized goals, and goal merging capabilities (each of which are important and much needed additions). The primary advantage of the approach Kaelbling takes is that reaction to external events is guaranteed to be bounded by a fixed delay between input and output. However, given that only limited run-time planning can be performed lacks flexibility. That is, though it appears that a GAPPS agent can accommodate external goals at run-time, there is no method of specifying new ways in achieving particular tasks without recompilation of the goal reduction rules and the circuits themselves.

The GAPPS system possesses the following REA characteristics.

Guaranteed Response The GAPPS system always generates an action in constant time. It may be possible to generate no–op actions in anticipation of forthcoming information, but responses are immediately guaranteed.

Failure Recovery Given that the entire system is compiled the system does not appear capable of recovering from failures. The GAPPS view is that there is no such thing as failure. That is, there are all sorts of things that can happen in the world and you should have a reaction for all of them. It is possible, however, to write a GAPPS program that does not account for every eventuality; in that case, it executes some programmer-specified default action.

Innate Behavior Assuming that the GAPPS agent was compiled with such abilities this capability would be provided.

Asynchronous Events The GAPPS/REX systems assumption is one of synchronous execution. But the cycle time should be fast (20 Hertz or so), and the agent "re-decides" on every cycle what it should be doing, so externally the behavior will appear as if it generates completely new actions in response to perceptual information. Nonetheless, the GAPPS agent does not possess this capability.

Weighing Alternatives There does not appear to be any weighing of alternatives in a GAPPS agent. It simply is composed of a finite set of condition–action pairs which allows it to respond to particular events without actually weighing alternative responses to an event. At compile time, you can, for instance, specify priorities on two different methods for achieving a particular goal. The resulting circuit chooses actions according to those priorities.

Change of Focus GAPPS is unable to change its computational focus. The circuitry always computes in a fixed manner. However, externally it may look like it's changing focus, because it may switch to entirely different perceptual and behavioral strategies based on inputs from the world. This means that focus is not a relevant issue for a GAPPS agent.

Predictability With the abilities to respond in a fixed interval between input and output and to guarantee that for a known input it will produce a response the GAPPS agent is said to behave predictably.

Temporal Reasoning It does not appear (according to [Kaelbling 88]) that temporal reasoning is performed in a GAPPS agent. However, if temporal reasoning is performed, it is definitely not done at run time.

2.2.4 Conditional Reaction to Observed Situations

Sanborn and Hendler have developed a dynamic reaction system called Conditional Reaction to Observed Situations (CROS) to study autonomous reaction in dynamic environments [Sanborn & Hendler 88]. The CROS approach is to use situation–driven reasoning about events taking place in a domain, as well as goal specifications from an offline planning system. Reaction in CROS is done by mapping discrepancy information onto action conditions to determine actions viable under a given prediction and potential failures.

The CROS system is composed primarily of two components — the State of Affairs (SoA) structure and functional objects known as monitors. The SoA is an extension of the notion of a plan state to incorporate change over time information for actions that conform to certain expectations. Change in a SoA is based on the interaction between an agent and its environment. Reaction is dependent on noticing discrepancies between the observable environment and a SoA which represents the expected unfolding of the near–term future as it affects an agent.

A monitor is responsible for noticing the discrepancies between the SoA and the environment by repeatedly testing a particular aspect of the environment against the SoA. There are four aspects to a CROS monitor: establishment, tracking, firing, and termination. When a monitor is established it determines what aspect of the environment it is responsible for and, thus supposed to track. It also determines the condition(s) under which it should notify the system by posting a change to the SoA. CROS uses discrepancy information sent by a monitor to enable, inhibit, or constrain actions, with some subsequent enabled action becoming the agents response to the discrepancy.

The CROS system possesses the following REA characteristics.

Guaranteed Response The CROS system does not synthesize plans at run time. It simply selects an appropriate response from those which it has indexed. This

facility suffices to provide this REA characteristic.

Failure Recovery It tries to predict failures via monitors and the SoA in order to prevent them from occurring. However, it does not appear to be able to recover from them once they occur.

Innate Behavior The system appears to be able to react without plans using its indexing scheme. Therefore, it appears that the system possesses this capability or is able to provide it.

Asynchronous Events This is provided through the use of monitors to a limited degree. That is, as long as a monitor exists to gather specific information then it can be used by the system. It does not, on the other hand, accept asynchronous events for which no monitor exists.

Weighing Alternatives The CROS system uses conditions of applicability and time on actions, as well as goal priority in selecting courses of action to pursue.

Change of Focus With the monitors making changes to the SoA the CROS system possesses the ability to change its focus of attention when discrepancies arise.

Predictability CROS does appear to perform predictably in that, it gives a particular response for a particular situation.

Temporal Reasoning It does not appear that this facility exists in CROS. While it is true that actions are selected on criticality, it appears that it is a factor of the action's duration and not a time-window in which the action must be performed. Nor does CROS provide any other temporal relations.

2.2.5 Reactive Action Packages

Jim Firby uses the concept of Reactive Action Package (RAP) in his execution system [Firby 89]. A RAP is an autonomous process which pursues a specific goal until that goal has been achieved, and an independent RAP exists for each goal in the system. A RAP consists of a partially ordered network of subtasks which are either a primitive command or a subgoal that will invoke another RAP.

Execution in the purely reactive RAP–based planner is according to the RAP interpreter which is loosely based on McDermott's NASL interpreter [McDermott 76]. The interpreter selects a RAP from its execution queue based upon temporal deadlines and on the ordering constraints on the subtask networks (called task nets). If the chosen RAP is a primitive command then it is passed to the hardware interface. Otherwise, the RAP is examined to see if its goal has already been accomplished. If it has been accomplished the RAP terminates successfully. If the goal has yet to be achieved, the RAP tries to select an appropriate task net which will achieve the goal. If one cannot be selected then the RAP fails. Otherwise, it sends the task net to the execution queue to run. When the RAP is selected from the queue again it either selects another task net until the goal has been achieved, or all task nets are ruled out and the RAP fails [Firby 87] .

The RAP–based planning system performs execution monitoring by requiring every action to return some form of feedback which it uses to update its world model. Firby also states that low–level replanning is performed. It appears that this replanning simply takes the form of re–execution of the failed command or task net or possibly by trying each task net in the RAP.

The RAP–based planning system possesses the following REA characteristics.

Guaranteed Response Since a RAP is made up of possibly applicable task networks it could select one without checking for its applicability which would allow it to possess this characteristic. However, it does not do this in the current implementation.

Failure Recovery When a failure of a RAP occurs the system continues to operate. The failure only implies that the particular goal which the RAP was supposed to achieve has not been achieved. When a failure occurs each method of the RAP is tried and failing that the RAP itself is retried to make sure the failure was not a simple anomaly of the execution environment.

Innate Behavior This capability is not provided in the RAP system. If the active and passive monitors available in the RAP system could be specified such that they were themselves considered by the interpreter and not only as the result of

another RAP, the RAP approach would possess the innate behavior characteristic.

Asynchronous Events Changes in the world model are brought about by the hardware interface in addition to the RAPs themselves. However, as primitive RAPs were executed with the TruckWorld simulator [Firby & Hanks 87] asynchronicity was not allowed since control was not given back to the RAP interpreter until the task had completed or failed. RAPs did not demonstrate asynchronicity, but the characteristic is present in essence (assuming asynchronous hardware).

Weighing Alternatives The interpreter selects RAPs based on approaching temporal deadlines and ordering constraints (along with some other heuristics). Therefore, this characteristic is present.

Change of Focus The RAP system is able to shift its focus of attention to more important tasks. If a task with higher priority is detected, the interpreter will switch its focus to the achievement of that task.

Predictability When a RAP finishes successfully, it is guaranteed to have satisfied its goal and to have executed all sensor action required to confirm that success.

Temporal Reasoning Temporal deadlines do exist between the subtasks of a RAP. However, they do not exist between RAPs, so the system weakly possesses this characteristic.

2.2.6 Entropy Reduction Engine

The Entropy Reduction Engine (ERE) is an architecture designed to integrate planning, scheduling, and control [Bresina & Drummond 90,Drummond & Bresina 90b]. The ERE architecture consists of three components: the reductor, the projector, and the reactor. The reductor's role is to transform a given behavioral constraint into a problem–solving strategy that is more effective at controlling the projector's search [Bresina *et al.* 93]. A behavioral constraint is based on a branching temporal logic and represents system behaviors (possibly with temporal extent). For example, a behavioral constraint may require that the distance to a particular object be precisely determined. Then with possible reductions of using cameras or sonar equipment the

ERE agent would have two alternative strategies to satisfy the constraint depending upon the situation, resource levels, cost, etc. The projector uses a causal theory and a behavioral constraint to produce situation–control rules (see [Drummond 89]). A causal theory is a set of operator schemata which defines both the actions that are controllable by the system and the exogenous events over which the system has no control. The causal theory is used to form a temporal projection which denotes possible future states and future actions. Stated another way, the projector simply searches through the space of possible event sequences using the behavioral constraint to limit that search. The reactor is basically a matching system which causes events to take place in the environment. The reactor accepts situation–control rules (SCR) from the projector and uses the current goal and facts describing the current environment to instantiate SCRs from all available ones. It then randomly selects a single SCR and an associated action to execute.

The ERE system possesses the following REA characteristics.

Guaranteed Response At startup the reactor is loaded with policies. These policies are actually SCRs which allow the system to function until better policies are developed by other parts of the system. This allow the system to guarantee a response though that response may not be the best possible response.

Failure Recovery ERE does not directly deal with recovery from failures. It does however, uniquely represent multiple outcomes of an action. Called stochastic outcomes, they try to anticipate the most probable outcomes for an action and if one of those outcomes occurs then the system can address it. On the other hand, if an outcome occurs which was not anticipated then the system fails. Therefore, ERE does possesses the capability to recover from failures though in a limited manner.

Innate Behavior In ERE the reactor can always do something even without a plan. It may be an incorrect action in the sense that it may cause problems for future deliberations, but it can nonetheless, react without a plan.

Asynchronous Events ERE has been developed with a sensor polling mechanism and does not accept asynchronous events from the environment. Without getting

into the debate as to whether polling at high frequency is comparable to accepting random events, ERE does not have the capability as defined.

Weighing Alternatives Even though SCRs are randomly selected in the reactor, as a system ERE does possess this capability in the projector.

Change of Focus With the ability to remove SCRs from the reactor, have new ones loaded, and later have the removed ones restored it can be said that ERE does have this capability.

Predictability ERE does appear to be predictable in its operation since it gives the same response for the same situation.

Temporal Reasoning Since behavioral constraints of maintenance and prevention are expressed over intervals of time, ERE does reason about goals with temporal extent.

2.2.7 Planner-Reactor

Lyons and Hendriks [Lyons 91,Lyons *et al.* 91] have developed a special purpose model of computation for representing highly conditional robot plans called RS which is an extension of the Robot Schemas model of Lyons and Arbib [Lyons & Arbib 89]. The P-R is an approach to integrating reaction and deliberation with the view that a planner is a system that continually modifies a concurrent and separate reactive system so its behavior becomes more goal directed [Lyons & Hendriks 92]. P-R is composed of two distinct and separate systems: the reactor and the planner. The reactor is initialized with useful behaviors which are encoded into reactions, cross-connected and annotated with time–constraints to allow for behavior without a specific plan. The planner takes a description of the environment, a description of the reactor, goal specifications, and constraints. It continually cycles making sure that the reactor is behaving in accordance with the user–specified goal specifications. If the reactor fails to meet those specifications the planner's job is to bring the reactor more in line with those specifications. In the RS model plans, actions and world models are represented as hierarchical networks of concurrent processes. This process representation allows behaviors to be conditional, concurrent, sequential, or iterative. An important contribution of their

work is how the planner is able to alter the structure of the reactor by specifying adaptations during execution.

The P-R system possesses the following REA characteristics.

Guaranteed Response Since the reactor is preloaded with default behaviors and continually scans its sensors, the system can guarantee some response for a given situation.

Failure Recovery The planner relies on "error detection" to improve the reactor—when some piece of the reactor fails it knows its time to replace it. However, it doesn't compute patches for erroneous plans, rather it computes generalizations of a reactor given that specific assumptions can no longer be taken. Thus, some limited form of failure recovery does take place.

Innate Behavior With preloaded behaviors the system does possess this capability.

Asynchronous Events Sensors are represented as processes that produce output, which can then be fed over interprocess communication links (called port-to-port connections) to other processes. Guardian processes can be set up to monitor sensors and report on events happening at (apriori) unknown times. Thus, the P-R system is capable of handling asynchronous events.

Weighing Alternatives Intuitively one would have to conclude that such a system must possess the ability to weigh alternate courses of action. However, it is not clear that this is the case for P-R from the references. Nonetheless, the system would not be able to function as it does without such a capability so we will assume that is possesses it.

Change of Focus It appears that the system could provide this ability with the planner adapting the reactor's behavior to address some new goal, and later restoring the old behavior.

Predictability The planner's input includes a set of assumptions that are likely hold in the environment, and the planner builts a reactor using all these assumptions. As it uses each assumption, it places 'guard' processes on the reactions that are

based on the assumptions. These reactions are then incrementally included into the reactor. If any of the assumptions used are then detectably false, the reactor will send perception messages back to the planner telling it so. The planner refines the reactor by building a better reactor in which these assumptions aren't used. Thus, the P-R system is able to function predictably.

Temporal Reasoning Temporal reasoning is a capability that the system possesses.

2.2.8 Task Control Architecture

Simmons uses a *structured control* approach by incrementally adding reactive behaviors to deliberative components in his Task Control Architecture (TCA) [Simmons 92]. Structured control involves the development of basic deliberative components that handle nominal situations and then increasing the reliability by incrementally layering on reactive behaviors to handle exceptions.

TCA is a high-level robot control operating system that provides an integrated set of commonly needed control constructs. It consists of modules that communicate by sending messages via a centralized control module. The central control module routes the messages to the appropriate deliberative components. Tasks are coordinated by specifying temporal constraints between nodes in the framework of hierarchical task trees [Simmons 90]. These constraints take the form of handling intervals, achievement intervals, planning intervals, sequential–achievement, and delay planning constraints. The central control is system/robot independent, while the other modules are task/robot specific.

Physical and computational resources are explicitly managed. TCA ensures that a resource's capacity is never exceeded, queuing messages if necessary until the resource becomes available, or locking the resource. This prevents other modules from accessing the resource until it is unlocked [Simmons 93].

Polling, interrupt–driven, and point monitors are used to detect unexpected changes in the environment. Polling and interrupt monitors operate concurrently with planned actions and test their conditions repeatedly for a specified duration or until a specified event occurs. Point monitors test conditions just once and are useful for determining

whether tasks have executed successfully.

To date, TCA has been used in over a half–dozen mobile robot systems [Simmons 93].

The TCA system possesses the following REA characteristics.

Guaranteed Response TCA is not able to guarantee a response in bounded time or without a task tree, mainly due to its use of the Ethernet for communications [Simmons 94].

Failure Recovery The system possesses fairly general capabilities for repairing and patching faulty plans when exceptional situations are detected. TCA provides facilities that enable modules to examine and modify task trees.

Innate Behavior Though it is not explicitly clear from the available literature that TCA does possess the ability for innate behavior it does appear possible within the design concept through the use of point, polling, and demon monitors.

Asynchronous Events The various monitor types of TCA allow it to detect and respond to asynchronous events.

Weighing Alternatives Task–dependent modules can weigh alternatives, and then use the TCA task tree modification techniques to prioritize the alternatives. Hence, the system does possess the weighing alternatives characteristic.

Change of Focus The system is able to change focus in the case of an exception, and through the use of monitors it is about to change focus to higher priority tasks.

Predictability One can use monitors and exception handlers to implement high–level servo/feedback loops whose behavior can be predicted, therefore TCA can be said to be predictable.

Temporal Reasoning Temporal constraints are allowed between any node in a task tree, no matter at which level. So, TCA possesses the temporal reasoning characteristic.

2.2.9 ATLANTIS

Gat, and many others, have adopted a three layered approach to designing execution systems that integrate planning and reacting specifically for the domain of robotics. His ATLANTIS architecture is a heterogeneous asynchronous architecture for controlling mobile robots based on a activity model of action.

The three layers of ATLANTIS are the deliberator, the sequencer, and the controller [Gat 91]. The deliberator is responsible for performing time consuming computational tasks such as planning, maintaining world models, and vision processing. These computations are initiated and terminated by the sequencer. The sequencer is responsible for controlling sequences of activities and deliberative computations. The sequencer is essentially Firby's RAP System [Firby 89] with the added abilities of simultaneously activities and preventing resource interactions between primitive tasks using semaphores. The controller is fundamentally a traditional analog feedback control mechanism whose transfer functions are written in Gat's language called ALFA.

The ATLANTIS action model is based on operators whose execution consumes negligible time, and thus do not themselves bring about changes in the world but instead initiate processes which then cause change [Gat 92].

The ATLANTIS system possesses the following REA characteristics.

Guaranteed Response The mechanism for controlling primitive activities computes a value for all inputs at every instant of time, thus the output may change, but there will always be an output. Additionally, the responses may be bounded by placing strict limitations on the lower and upper bounds on transfer functions, so this characteristic is possessed by the system.

Failure Recovery The failure recovery capabilities are essentially the same as the Firby's RAP system [Firby 89] with the added ability of clean–up procedures. Thus, the system does possess the capability to address failures.

Innate Behavior The ATLANTIS system can operate without its sequencing and deliberative layers, therefore the non–linear, stateless transfer functions of the control layer possess the ability to allow the system to operate without a plan.

The control layer is implemented in ALFA and is similar to the circuit semantics of REX [Kaelbling 88], thus responses can be generated independent from a plan as long as the stimuli are appropriate for the hardwired circuit.

Asynchronous Events One of the design goals for ATLANTIS was that it handle asynchronous events since it was to control robots. The system is able to respond to both contingencies and opportunities and, thus possesses this characteristic.

Weighing Alternatives Since the sequencing layer is responsible for initiating activities, and it is based upon Firby's RAP system, ATLANTIS weighs alternatives in the same manner as the RAP interpreter.

Change of Focus Again, with the sequencing layer being essentially Firby's RAP interpreter, the system is able to change its focus of attention in the same way the RAP system does.

Predictability With the control layer being a set of circuits responding to inputs from the environment and activations from the sequencing layer, the system will perform predictably.

Temporal Reasoning The present ATLANTIS system was not designed to include a temporal reasoning capability, however it appears that adding one to the deliberative layer would be possible within the design concept.

There are many systems that also could be included in this sample. These include the work by [Brooks 86], [Agre & Chapman 87], [Connell 92], [Downs & Reichgelt 92], [Musliner *et al.* 93], and [Bonasso & Kortenkamp 94] to name a few. However, the purpose was not to exhaustively examine the literature (though this is a worthy goal), but to examine a diverse, broadly representative sample, that covered the approaches being used to date.

2.3 Characterization Summary

This comparison classifies each of the systems in the sample according to how well they implement a particular characteristic or provide the underlying mechanisms to allow the addition of the characteristic to their design.

The classification (see Table 2.1) is made along the lines of a "three-star" rating system. Three stars indicate that the characteristic is implemented so that the definition (Section 2.1) is exactly satisfied. Two stars imply that the characteristic is implemented, but its implementation does not fully satisfy the definition. A single star indicates that the characteristic is not implemented, yet there appears to be the necessary underlying structure to implement the capability. If no star is indicated then the particular system neither implements the characteristic nor appears to have enough underlying structure for the characteristic to be implemented. A characterization of system capabilities such

	PLANEX	PRS	GAPPS	CROS	RAP	ERE	P-R	TCA	ATLANTIS
Guaranteed Response		***	***	**	*	***	***	*	***
Failure Recovery	**	**			***	**	**	***	***
Innate Behavior			*	**	*	***	***	*	**
Async. Events		***		**	**	*	**	***	***
Weighing Alternatives		**	**	**	***	**	*	***	***
Change of Focus		***		**	***	*	*	***	***
Predictable	***	***	***	***	***	***	***	***	***
Temporal Reasoning		**			**	**	**	***	*
	1972	1987	1988	1988	1989	1990	1991	1992	1992

*** Strong support for feature
** Weak support for feature
* Could be added within design concept
- Not apparent in design concept

Table 2.1: Characterization of Rational Systems

as the one presented in Section 2.2 is a difficult task. One problem is that even though a system exhibits a particular REA characteristic that fact alone does not mean that the system possess the characteristic to the same exact degree as another system. For example, the abilities of handling asynchronous events in PRS are more advanced to those of CROS yet both provide a facility to handle such events. Some characteristics are readily apparent in systems while not so clear in others, but with some thought one can visualize how a particular characteristic might be added to the design[2].

There are many characteristics that a system may possess and different designers place

[2] This assumes that there is enough underlying structure to support the functionality required of the characteristic—shown as a single star in Table 2.1.

different emphasis in the design on any one for a variety of reasons (e.g., research interest, time and/or funding constraints, etc.). The purpose of this characterization has been to examine the primary systems that stand out in the AI literature to determine if those systems possessed similar characteristics. It is not the purpose to present one particular system as being superior to another. We won't be able to do that sort of comparison until someone develops a set of benchmarks by which all systems can be fairly evaluated, such as those called for in [Drummond & Kaelbling 90].

What can be determined from such a characterization is whether the characteristics defined here are representative of the capabilities necessary for rational behavior in reactive systems. As can be seen from Table 2.1 the characteristics identified in Section 2.1 are exhibited by most systems developed over the past few years. However, simply identifying those characteristics is not enough. As Georgeff points out [Georgeff 90], the decisions of how these capabilities are to interact to provide rational behavior is based upon a variety of factors. These include the likelihood of success of the task, its utility, resource requirements, expected execution time, information availability, information reliability, and commitments on other tasks. Thus, it's simply not a matter of making a decision, but rather making a decision which has particular trade–offs that make it the "best" decision for the moment. Just how this is actually done is not understood, but forms the motivation for the design presented in this research.

This is definitely not the first attempt to characterize some aspects of reactive systems. The interested reader is directed to the following works. Kaelbling [Kaelbling 90] examines the methods for specifying behaviors in agents. Lyons and Hendriks [Lyons & Hendriks 92] provide an excellent overview of the reactive planning literature. Hanks and Firby [Hanks 90] describe the issues in acting versus deliberation, and Drummond and Kaelbling [Drummond & Kaelbling 90] call for benchmark and evaluation metrics. Also, see McDermott's [McDermott 92] work for some more interesting approaches to addressing some of the open research issues.

2.4 MAD - Modeling

The basis for the REA design is the model of a rational agent as identified by the characterized capabilities. These are:

- the ability to guarantee a response in bounded time;
- the facilities to recover from execution failures and continue to operate;
- default innate behaviors which allow the agent to survive in a dynamic environment and act without having to deliberate;
- acceptance of asynchronous events to react to exogenous occurrences and changes in the environment;
- the weighing of alternative courses of action to choose actions which are most appropriate for the environmental circumstances;
- mechanisms to change the focus of attention to address more critical tasks;
- predictable behavior; and
- facilities for reasoning about time.

The success of the design to produce an agent which behaves rationally while situated in a dynamic environment must, at a minimum, be measured against these characteristics of the model. Thus, using the metric of the "star" rating (adopted from the characterization), the design should yield an agent that has a rating of "***" for each of the characteristics. This will be measured by the demonstration of the REA design in the Pacifica domain if all of the following aspects can be shown to be present for each of the characteristics.

Guaranteed Response	(1) Act with a single task directive. (2) Act without a task directive (i.e., using innate behavior). (3) Act with multiple task directives.
Failure Recovery	(1) Recover from an exogenous event which the REA can address through procedural knowledge.

	(2) Detect an early failure and request assistance from the superior agent. (3) Detect and recover from a failure detected by behavior mechanism. (4) Detect an exogenous event which the REA cannot address and request assistance from the superior agent. (5) Detect a precondition failure at action execution time.
Innate Behavior	(1) Act without a task directive performing some internal maintenance action. (2) Act without a task directive in response to some external stimulus.
Asynchronous Events	(1) Accept communication from the superior agent while processing one or more task directives. (2) Accept event from the environment while processing one or more task directives.
Weighing Alternatives	(1) Selection of a task procedure when several are present. (2) Selection of schema when several are present. (3) Selection of a task directive to process when several are present.
Change of Focus	(1) Detection of an event by a behavior which causes the introduction of procedural knowledge to address the event. (2) Acceptance of a task directive with a higher level of commitment than a directive which is presently being processed causing the new directive to be executed and the old one to wait.
Predictability	(1) Deterministic behavior in different environments under similar circumstances.
Temporal Reasoning	(1) Acceptance and abeyance of task directives with temporal deadlines.

(2) Abeyance of temporal constraints on actions of task directives.

2.5 Chapter Summary

This chapter has identified a set of basic characteristics necessary for the design of rational agents that are to be situated in dynamic environments. Those characteristics are that the agent must:

- be able to guarantee a response in bounded time;
- possess facilities to recover from execution failures and continue to operate;
- have default behaviors which allow the agent to survive in a dynamic environment and act without having to deliberate;
- accept asynchronous events to react to exogenous occurrences and changes in the environment;
- weigh alternative courses of action to choose actions which are most appropriate for the environmental circumstances;
- have mechanisms to change the focus of attention to address more critical tasks;
- be predictable in its behavior; and
- possess facilities for reasoning about time.

Investigating the problems of integrating together the approaches taken in other systems allows us to develop guidelines for designing rational agents. By using such guidelines we will be able to design agents which are both more rational and practical than those which exist today.

In the next chapter we discuss how the REA we build from this characterization of rationality is able to reason about its environment and the tasks under its control.

Chapter 3

World Model and Task Behavior Language

In Chapter 2, we saw that a reactive agent which is situated in a dynamic environment can be characterized by a basic set of capabilities. However, some additional functionality is required to allow these capabilities to be integrated into a reactive system. We still require a means to reason about the environment and to assimilate information from the environment upon which to base decisions. That is, we need a mechanism to model entities in the environment under the control of the REA, and things which may affect those controlled entities. In addition, we require a means of specifying how tasks to be carried out in the environment will behave. We require a means to describe aspects of a task's behavior and the ways of achieving the task's desired outcome in a variety of situations.

This chapter discusses the organization of the REA's database which is used to model its environment, and the language used by the REA for specifying the behavior of tasks at execution time. The first section describes the functionality of the World Model, how sensor information is assimilated into the model, and the interface to retrieve information stored in the model. The next section describes the syntax and semantics of the REA's language for specifying task execution behavior.

3.1 World Model

The best way to model the world would be not to model it at all (*cf.* [Brooks 86]) Ideally, we could directly sense any necessary information, thus yielding a more accurate description than could be gained with a database of stored facts. The problem with this approach is threefold. First, sensing rarely comes at zero cost. Making sensor requests takes time and potentially utilizes resources which have limited availability. Therefore, such an approach would be very costly. Second, sensors only provide information about the immediate vicinity of the agent. This constraint limits the types of actions which could be chosen. For example, the agent would not be able to choose a road for traveling to a particular destination if only local information existed. Lastly, a purely sensor–based approach would not provide an agent with a means of storing and reasoning about unsolicited information: that is, information which is available as a result of execution or failure of a task, and not directly from an explicit sensor request. A purely sensor–based approach would thus limit the types of domains in which an agent could be situated, and in which it could be said that it performed rationally.

Acting intelligently in an environment requires that a REA maintain some sort of dynamic model of that environment. The agent must be able to model those aspects of the environment necessary to decide its best course of action given its knowledge of its environment, and those aspects which allow it to detect failures—failures which result from its own actions as well as those of other agents in the environment. Therefore a world model is a database of beliefs about the world in which information regarding modeled aspects of the environment is stored once observed and persist until new information is acquired through sensors or by unsolicited reports from the environment.

There are a variety of issues which must be addressed in the design of a world model for an execution system. These include the type of data to be maintained (i.e., only facts, or a combination of facts and control information), how that data will be stored and retrieved, the form in which the data will be acquired, whether to allow additional data to be derived from data contained in the model, and whether mechanisms (other than the World Model) of the agent can alter the data in the model.

The approach taken was to develop a model which contained only beliefs held about

specific modeled entities in the environment. The model would have fast and efficient mechanisms for storing and retrieving data, allow for limited forward chaining of particular types of data, and be globally accessible in the architecture so all components could have access to the available knowledge of the environment. The design of this model is discussed in the next section.

The model does not address the issues associated with indexical/functional reference models [Chapman & Agre 86,Schoppers & Shu 90] and assumes that there exists an interface which converts sensor data into predicates corresponding to particular modeled entities in the environment. These issues remain open research topics. However, Firby [Firby 89] began to address these issues by assigning a temporary name to each individual item encountered in the world and using that name as long as the item remained in sensor range. When the item was out of sensor range the name was lost and a rather involved process had to take place to determine if a new item was in fact one that had been identified before. The complexity involved in matching item descriptions as Firby did is not necessary for this discussion, and it is assumed that such a capability exists.

3.1.1 Model Design

The underlying mechanism of the World Model is a discrimination network [Charniak *et al.* 87] integrated with unification and pattern matching abilities. The resulting model allows both the keys and queries to be predicate structures that may contain variables.

Queries and assertions are expressed in the same language[1]. There are two types of objects allowed in the language: *categories* and *individual.* In a predicate calculus sense, a category corresponds to a one–place predicate and an individual to a constant. Statements in the language must have one of five operators: *sub*, *ind*, *val*, *and*, and *or*. Their general and formal forms are:

[1] Adopted from [Norvig 92].

(**sub** *subcategory supercategory*)	$\forall x : A(x) \supset B(x)$
(**ind** *individual category*)	C(I)
(**val** *property individual value*)	P(I,V)
(**and** *association...*)	A $\wedge$ B...
(**or** *association...*)	A $\vee$ B...

Examples of assertions in the World Model are of the form:

(ind gt1 ground-transport)
(val at gt1 city-k)

which states that gt1 is an object of type ground-transport, and that gt1's *at* property has the value of city-k (i.e., gt1 is presently located at city-k).

Such assertions are made as a result of receiving either unsolicited information, or information from an explicit sensor request. Unsolicited information comes from two sources—task initiation or failure, and task completion. When a task is dispatched to the environment it may immediately return information on the state of some entity in the world as an effect of beginning a particular task, or upon failure to complete a task. This is unsolicited task initiation or failure information. When a task completes it returns information specific to the type of task which completed such as, status of resources used, etc. From the standpoint of the REA these sources are treated in the same manner and no distinction is made. The REA receives such information asynchronously and it assimilates this new information by asserting it into its World Model so that appropriate actions can take place. As sensor data is received, the REA determines the type of sensor returning the information and selects a corresponding model of data from that sensor to assimilate only information that it is concerned about. Chapter 7 describes this in more detail. As assertions are made from these sources, previous assertions for particular properties are removed from the World Model. These properties only have one value at a time.

3.1.2 Queries

Queries are executed by matching predicate structures, which may contain variables, against assertions in the World Model. No backward–chaining is allowed in the world model; therefore, no deduction can take place. Query satisfaction is simply a matter of unifying a formula with assertions. Allowing deductions to be made would violate the

design goal of having a fast retrieval mechanism since deductions could take arbitrary amounts of time before choices are returned.

The query mechanism has a functional interface which returns one of two types of value. The first type is a single value which is the most recent value of a particular property for an individual. This interface has the general form:

(*property individual*)

For example,

(res-status 'c5-1)

returns the current status of the c5-1 resource. The second type of value is a binding list which is all values which existentially match variables in the query. This interface has the general form:

(*property individual* | <*ind-var*> [*value* | <*val-var*>])

<ind-var> ::= ?<name>
<val-var> ::= ?<name>

Examples are:

```
(res-status 'gt2 '?what)
(at '?resource 'delta)
(at '?resource '?where)
```

where the first query returns the binding list `((?what . available))` that states that the only possible binding of the variable "?what" is "available." The second query returns the binding list `((?resource . c5-1)) ((?resource . b707))` stating that the resources currently located at Delta are the b707 and c5-1. The third query returns a binding list stating all of the possible bindings for any resource located anywhere.

Predicate structures may be combined using the logical connectives *and*, *or*, and the unary connective *not*. Additionally, numeric and relational functions such as, *+*, *-*, ***, */*, *=*, *<*, *<=*, *>*, *>=*, *eq* and *equal* may be used to test for specific values. For example:

```
(and (at 'gt1 '?where)
     (weather '?where '?what)
     (> (fuel 'gt1) 10)
     (<= '?what 4))

(or (and (res-status 'gt1 'available)
         (res-status 'gt2 'unavailable))
    (and (res-status 'gt1 'unavailable)
         (access 'road-ab 'closed))
    (and (not (res-status 'gt1 'available))
         (at 'c5-1 'edinburgh)
         (= 0 (nationals 'delta))))
```

3.1.3 Sensor Models

Sensors often return much more information than is needed by the REA; information such as internal status of resources which the REA does not use for making decisions or has no need to know. Therefore, each type of sensor known to exist in the environment has a data model. The model maps the data registers of the sensor to property predicates in the World Model. Only properties which can dynamically take on new values are asserted in the World Model. Static information which the sensor may report is not asserted since it has either already been asserted or it is not something that decisions will be based upon. This approach reduces the number of assertions made to the World Model and avoids the unrealistic assumption that sensors are returning predicates directly from the environment to the REA.

3.1.4 Fact Persistence

Though particular properties of an entity in the environment have values in the World Model we cannot believe that those values will hold forever. As time continues to elapse the *belief* which we have for property values should decay. We cannot expect that if we observe a spider climbing down a wall, that when looking for the spider again in two days that we will find in the same place. Thus, we need some method of knowing how much faith to put in a value which is retrieved from out model of the world. The approach taken is to assign to each assertion a timestamp at the time it is asserted (*cf.* [Firby 89]).

A timestamp is given as a single non–negative integer[2], and is automatically assigned to each assertion. The general form of the timestamp construct is:

(**time–stamp** *predicate–structure time*)

where *predicate–structure* is a simple predicate structure with no connectives or boolean functions and *time* is a non–negative integer. This construct is used to determine the validity of information contained in the World Model. For example, if we wanted to determine the last time the act-sensor was used to report information about the c5-1 resource we could use:

```
(time-stamp '(sensor act-sensor c5-1) '?time)
```

which would return a binding list with the potential bindings of the variable "?time" given. We could combine this query to determine whether the value of "?time" had some temporal relation by the following query.

```
(and (time-stamp '(sensor act-sensor c5-1) '?time)
     (<= (* (- (get-universal-time) '?time) 60)
         *act-sensor-freq*))
```

which would inform us about the last time the act-sensor was used and whether that time was before or met the next usage interval.

An alternative approach would be to use probabilities to determine which propositions hold at a specified point in time [Hanks 92]. Such an approach however, is dependent upon having an active model of the world that assigns probabilities that vary with time and related facts. That is, a probabilistic model would have to keep track of contextual dependencies between event types and calculate the likelihood of such events occurring. There are advantages to both approaches, but by making the user of the knowledge from the World Model responsible for determining the validity in a local context the timestamp approach is more attractive as it reduces the required model complexity.

---

[2] This integer, known as Universal Time, is the number of seconds since midnight, January 1, 1900 GMT.

3.2 Task Behavior Language

An underlying premise of an execution system charged with controlling entities in a dynamic environment on behalf of another is that it be taskable. That is, it should be possible to specify tasks which the agent is to carry out and which tasks the agent is to abandon. The agent must choose appropriate tasks for the situation in which it finds itself that allow it to achieve those tasks it intends to carry out. The execution of tasks should allow the agent to behave in a prescribed manner which is both rational and robust in the face of unexpected change. Thus, we require a language that will allow the agent to represent tasks and allow us to specify the behavior of those tasks for different situations.

Drawing upon the success of behavior–based control systems and overcoming their limitations, Firby developed a representation of Reactive Action Packages, or RAPs, to provide a description for the execution of tasks [Firby 92]. The Task Behavior Language (TBL) presented in the following sections is derived from the RAP approach to representing tasks. The TBL adds additional constructs to the RAP language to make it more akin to planning operators and to provide additional functionality, and changes the syntax of some of the constructs. Some terminological changes have been made as well. TBL has been developed from scratch, but the reader should be aware that the RAP approach heavily influenced the direction taken. For further details of the RAP language the reader is referred to [Firby 89].

The TBL is an object oriented, hierarchical language with five primary object constructs: task directive, task schema, domain data, monitor, and behavior. Task directive, schema, and monitor objects are dynamically created at execution time by the REA. Domain Data Objects (DDOs) and behavior objects are either defined by the user and loaded from a library when the REA initializes or are added at runtime via message types of the Inter–Agent Communication Language (Chapter 5). The library approach allows the planning agent to represent tasks at a greater level of abstraction since the REA is able to represent the detailed steps that are necessary to execute a task in the environment. This library represents the REA's knowledge of tasks it can perform in the environment, and each DDO serves as an abstract planning operator

out of which plans can be assembled. The use of a library addresses the problem of choosing appropriate actions under stringent time constraints [Hanks 90].

In the following sections we shall examine each of the object types of the TBL and discuss the meaning of each of their associated attributes.

3.2.1 Task–Directive Object

When the REA is tasked (i.e., commanded) to execute actions in the environment by a planning agent a task–directive object is synthesized. The task–directive object contains the information specified in an IACL *synthesize* message (Section 5.2.7) along with information necessary for the REA to process and reason about the tasks contained in the task directive.

```
#<Task-Directive #XE4C90E>
    is an instance of the class TASK-DIRECTIVE:
 The following slots have allocation :INSTANCE:
 NAME                    NIL
 E-STATUS                NIL
 PRIORITY                NIL
 TASK                    NIL
 PROCESSED               NIL
 PROCESSING              NIL
 ORDERING-CONSTRAINTS    NIL
 CSTR-TABLE              NIL
```

Figure 3.1: Uninstantiated Task Directive Object

The task-directive object contains information regarding the execution status of the directive, the priority of the directive, the tasks which make up the directive, ordering constraints between the tasks of the directive, causal structure of the tasks, processing and processed tasks, and a reference table which allows the REA to communicate information about tasks in a form understood by the planning agent. This section describes the purpose of each slot of the task–directive object.

Name and E–Status

The name provides a unique identifier for the Task Directive Object.

The execution status, or e–status, of a task directive can be instantiated with one of three values at any give time: **U**nscheduled, **R**eady-for-processing, or **P**rocessing. These values inform the control mechanism of the REA as to the state of execution of the task–directive, and allow for appropriate processing to be conducted. See Chapter 4 for a detailed discussion on how these values are used during execution of a task–directive object.

Priority

The commitment of the REA to a particular task–directive is determined by that task–directive's priority. This value, specified by a planning agent, informs the REA how to choose between directives when such choices must be made.

Task

The REA reasons about the tasks of the directive using these objects. When a task–directive object is synthesized from information provided by the planning agent (see Section 5.3) in an IACL *synthesize* message, the node–network information is used to create task–schema objects for each node in the network. Once created, these schema objects are stored in the task slot of the task–directive object.

Processed and Processing

The processed and processing slots of a task–directive object are used during the execution of the directive to allow the REA to reason about which tasks have been executed, which have yet to be executed, and which are presently processing.

Ordering–Constraints

During the generative planning process, ordering constraints can be imposed between actions to reduce the chances that conflict will occur. This information is provided in an IACL *synthesize* message so that the REA can properly order the execution of tasks for a task–directive. However, the ordering constraints are not necessarily linear, and may indicate that tasks are to be executed in parallel. The REA uses this information when deciding which task(s) to execute next.

Causal Structure Record Table (CSTR)

The causal structure of a plan states the relationship between the purposes of actions with respect to the goals or sub–goals they achieve for some later point in the plan [Tate 77]. This information is communicated to the REA in an IACL *synthesize* message to provide a means to detect, as early as possible, when something has gone wrong during execution. The REA uses this information to create monitor objects which monitor specific aspects of the execution to determine if assumptions have been violated. The causal–table slot maintains this causal structure information.

3.2.2 Domain Data Object

During the task–directive synthesis process, task schema objects are created according to node–network information specified in an IACL *synthesize* message. These task schema objects are constructed from a user–defined library of Domain–Data Objects (DDOs). Each DDO contains information which specifies the execution–time behavior that a specific task (that the REA has the ability to carry out) will have in the environment.

A DDO contains required and optional information which the REA uses to execute a task. These include the pattern which identifies the task, preconditions on the applicability of the task, the procedures which describe the task's behavior, the effects which can be expected to be brought about by executing the task in the environment, domain–specific calculation functions, information on resources required by the task, a heuristic estimation of execution cost, the condition which specifies when execution

```
#<Domain-Data #XE40F06>
    is an instance of the class DOMAIN-DATA:
The following slots have allocation :INSTANCE:
NAME                        NIL
EXPANDS                     NIL
CONDITIONS                  (QUOTE (ALWAYS-TRUE))
CALC                        NIL
REPEAT-WHILE                NIL
PROCEDURES                  NIL
EFFECTS                     NIL
USES-RESOURCES              NIL
EXEC-COST                   0
END-COND                    NIL
DURATION                    (0 0)
```

Figure 3.2: Uninstantiated Domain Data Object

has successfully completed, and an execution duration estimate. This section describes the purpose of each slot of the DDO.

Name and Expands

The name provides a unique identifier for the Domain–Data Object.

The pattern specified in the expands slot is used by the REA to identify which DDOs are appropriate to a particular task. The general form of the expands slot is:

$$\text{:expands '}(name\ [v_1, \ldots, v_n]^*)$$

where *name* is unique to a specific task[3], and $[v_1, \ldots, v_n]^*$ specifies zero or more fully instantiated input variables.

For example, typical patterns of expands slots are:

```
:expands '(fly-to-dest ?res ?from-loc ?to-loc)
:expands '(refuel ?res)
:expands '(update-model)
```

[3] The *name* specifier is unique to a particular task, but this does not mean that multiple DDO expand slots cannot have the same pattern. This is allowed to provide the user with the ability to specify multiple methods of achieving a particular task with different execution costs.

The first pattern is for the task fly-to-dest with three variables, the second is for the task refuel with one variable, and the third is for the task update-model with no variables. When schema objects are to be created according to network–node information the REA searches the library of DDOs looking for all candidates which unify the expands pattern for a particular task.

Conditions

DDO conditions specify what must be satisfied according to the REA's model of the world before any procedure of the DDO can be executed.

This information along with that provided by DDO effects liken the schema object that is created from the DDO to a classical planning operator. Specification of DDO conditions is optional.

The general form[4] of the conditions slot is:

:conditions '<statement>

```
<statement>      ::= (<L-connective> [<stmt>]+) |
                     (not <statement>) |
                     <clause>
<stmt>           ::= <clause> | <statement>
<L-connective>   ::= and | or
<N-connective>   ::= + | - | * | / | =
<R-connective>   ::= <= | >= | eq | equal
<clause>         ::= <ground> |
                     (<N-connective> <expr> <expr>) |
                     (<R-connective> <expr> <expr>)
<expr>           ::= <clause> | <variable>
<ground>         ::= (property entity value) |
                     (property entity <variable>) |
                     (property <variable> value) |
                     (property <variable> <variable>) |
                     value
<variable>       ::= ?<name>
```

A clause may contain uninstantiated variables which become temporarily bound to test the validity of the clause. These bindings remain for testing the remaining conditions

[4] The BNF notation given in the general form assumes the usual semantic conventions.

and may be used as values in subsequent condition clauses. However, these bindings are lost once the conditions have been tested. Example conditions are:

```
:conditions '(and (or (resource-status '?resource 'available)
                      (resource-status '?resource 'boarded))
                  (at '?resource '?from-location))

:conditions '(and (fuel-level '?resource '?gallons)
                  (max-fuel '?resource '?max)
                  (<= '?gallons '?max))
```

The first condition states that the resource denoted by the variable *?resource* be available or boarded, and at the location denoted by the variable *?from-location.* The second states that the fuel level of *?resource* be some number of *?gallons* and that the number of gallons be less than or equal to the maximum number of gallons for *?resource.*

Calc

The calc slot of a DDO provides a mechanism by which the schema created from the DDO can acquire information at execution time. This mechanism uses a functional interface to the REA's World Model to perform analysis of information in the model that is obtained by a more complex means than that of simple queries. The specification of the calc slot is optional.

The general form of the calc slot is:

:calc '([analysis-function [<inputs>]* binding-variable]*)

<inputs> ::= <query> | <variable>
<query> ::= (property entity)
<variable> ::= ?<name>

The result of the *analysis–function* is bound to the *?binding–variable.* The ?binding–variable from each analysis–function call is then available for use in the procedures of the DDO. An example of the use of the calc slot is found in Figure 3.3. The drive DDO calc slot uses two analysis functions called set–route and set–speed. Set–route is

defined to calculate the shortest route between two locations taking into consideration road access, road conditions, and distance. Set-speed calculates the maximum safe speed a ground transport can travel taking into consideration road conditions, the

```
(make-instance 'domain-data
     :name 'drive
     :variables '(?res ?to-loc ?from-loc)
     :expands '(drive ?res ?to-loc ?from-loc)
     :conditions '(and (or (res-status '?res 'available)
                           (res-status '?res 'driving)
                           (load-status '?res 'loaded))
                       (at '?res '?from-loc))
     :calc '((set-route (at '?res) '?to-loc '?road)
             (set-speed '?res '?road (at '?res) '?to-loc '?speed))
     :repeat-while '(and (at '?res '?location)
                         (not (eq '?location '?to-loc)))
     :procedures
      (list
        '(DRIVE-1 network
           (context (and (fuel-level '?res '?gallons)
                         (max-fuel '?res '?max)
                         (> '?gallons (* *ground-threshold* '?max))))
           (network ((tn1 (go-location ?res ?from-loc
                                ?to-loc ?road ?speed) () ())
                     (tn2 (city-sensor ?location) () ())))
           (orderings ((tn1 nil (tn2))
                       (tn2 (tn1) nil)))
        ...)
     :effects '((at '?res '?to-loc))
     ...)
```

Figure 3.3: Portion of the Drive Domain–Data Object from Pacifica

type of road surface, weather, and the mechanical status of the vehicle. Each time the drive DDO comes up for execution the analysis functions calculate their respective values and store them in the variables *?road* and *?speed*[5]. This information can then be used to select between procedures, or in the case of the drive DDO, simply provide additional specifications for the execution of a task before it is dispatched.

[5] Though the analysis functions use information stored in the World Model for their calculations, the results are used only within the scope of the DDO and are not asserted into the World Model.

Repeat–While

To allow explicit specification of repetition or looping behavior during execution the repeat–while slot is provided. The statement specified in the repeat–while slot of the DDO conforms to the same requirement and restrictions as does the conditions slot. The difference is the repeat–while statement is not used to filter the selection of the schema created from the DDO, but rather to make sure that certain conditions are present before execution can continue. That is, if the repeat–while statement is satisfied according to the REA's World Model then the control mechanism is free to select any of the schema's procedures to continue execution to bring about the desired effects. The specification of the repeat–while slot is optional.

The general form of the repeat–while slot is:

:repeat–while '<statement>

<statement>	::= (<L-connective> [<stmt>]$^{+}$) |
	(**not** <statement>) |
	<clause>
<stmt>	::= <clause> | <statement>
<L-connective>	::= and | or
<N-connective>	::= + | - | * | / | =
<R-connective>	::= <= | >= | eq | equal
<clause>	::= <ground> |
	(<N-connective> <expr> <expr>) |
	(<R-connective> <expr> <expr>)
<expr>	::= <clause> | <variable>
<ground>	::= (property entity value) |
	(property entity <variable>) |
	(property <variable> value) |
	(property <variable> <variable>) |
	value
<variable>	::= ?<name>

For example, consider the following repeat–while statement from the drive DDO in Figure 3.3.

```
:repeat-while '(and (at '?resource '?location)
                    (not (eq '?location '?to-loc)))
```

This states that while the ground transport bound to *?resource* is at a location, and that location is not the destination of the drive task then it is okay to continue with the task. What actually happens for the drive DDO is that as the ground transport arrives in a new location it informs the REA. The REA checks to see if the drive DDO has been satisfied (i.e., has reached its destination). If not, then since the drive DDO is repeatable and its conditions for repetition are satisfied, the analysis functions of the calc slot are recalculated and the drive task is dispatched again. The result is to send new route and speed information to the ground transport so that it can reach its destination from its present location.

Procedures

A task's execution time behavior is defined by the contextually dependent procedures specified in the DDO. A DDO may have an arbitrary number of procedures that define how the task can be achieved in a number of situations.

There are two types of procedure namely, a *network* procedure or a *primitive* procedure. A network procedure defines a task which is composed of sub–tasks. For example, the ftd-1 procedure of the fly-to-dest DDO (shown in Figure 3.4) is a network procedure that states that the fly-to-destination task involves taxiing the plane to the runway, requesting and receiving clearance, flying the plane to its destination, and landing the plane at its destination location. A primitive procedure defines a task which is composed of a single atomic action. For example, the taxi-1 procedure of the taxi DDO (shown in Figure 3.5) is a primitive procedure that can be directly executed in the environment. The difference between the two types of procedure is that *network* procedures must first be decomposed to primitive procedures, while *primitive* procedures require no decomposition before they can be executed. The *context* is used as a query to the World Model to determine if conditions are satisfied for a particular procedure to be used.

Each procedure contains a *context* and a *network* or *primitive*. The context defines the situation in which the procedure is appropriate and the network or primitive defines the behaviors or behavior that achieves the task.

```
(make-instance 'domain-data
     :name 'fly-to-dest
     :variables '(?res ?from-loc ?to-loc)
     :expands '(fly-to-dest ?res ?from-loc ?to-loc)
     :conditions '(and (or (res-status '?res 'available)
                           (res-status '?res 'boarded))
                       (at '?res '?from-loc))
     :procedures
      (list
        '(FTD-1 network
           (context (and (fuel-level '?res '?gallons)
                         (max-fuel '?res '?max)
                         (>= `?gallons (* *air-threshold* '?max))))
           (network ((tn1 (taxi ?res) () ())
                     (tn2 (get-clearance ?res) () ())
                     (tn3 (liftoff-fly ?res ?from-loc ?to-loc) () ())
                     (tn4 (land ?res ?to-loc) () ())))
           (ordering ((tn1 nil (tn2))
                      (tn2 (tn1) (tn3))
                      (tn3 (tn2) (tn4))
                      (tn4 (tn3) nil))))
        '(FTD-2 network
           (context (and (fuel-level '?res '?gallons)
                         (max-fuel '?res '?max)
                         (< `?gallons (* *air-threshold* '?max))))
           (network ((tn1 (refuel ?res) () ())
                     (tn2 (taxi ?res) () ())
                     (tn3 (get-clearance ?res) () ())
                     (tn4 (liftoff-fly ?res ?from-loc ?to-loc) () ())
                     (tn5 (land ?res ?to-loc) () ())))
           (ordering ((tn1 nil (tn2))
                      (tn2 (tn1) (tn3))
                      (tn3 (tn2) (tn4))
                      (tn4 (tn3) (tn5))
                      (tn5 (tn4) nil)))))
     :effects '((at '?res '?to-loc)
                (parked-at-gate '?res 'yes))
     :uses-resources (list '?res)
     :exec-cost 3
     :end-cond '(and (at '?res '?to-loc)
                     (parked-at-gate '?res 'yes))
     :duration '(81 127))
```

Figure 3.4: Fly-to-dest Domain–Data Object from Pacifica

The general form of the procedures slot is:

```
:procedures ('(procedure-name <procedure-type>
                    (context <statement>)
                    (network ([<net-spec>]+))
                    [(orderings ([tag (pre-nodes) (post-nodes)]+))]*))
```

```
<procedure-type>   ::= network | primitive
<statement>        ::= (<L-connective> [<stmt>]+) |
                       (not <statement>) |
                       <clause>
<stmt>             ::= <clause> | <statement>
<L-connective>     ::= and | or
<N-connective>     ::= + | - | * | / | =
<R-connective>     ::= <= | >= | eq | equal
<clause>           ::= <ground> |
                       (<N-connective> <expr> <expr>) |
                       (<R-connective> <expr> <expr>)
<expr>             ::= <clause> | <variable>
<ground>           ::= (property entity value) |
                       (property entity <variable>) |
                       (property <variable> value) |
                       (property <variable> <variable>) |
                       value
<net-spec>         ::= (<tag> <pattern> ([<temporal>]*) ([<preference>]*)) |
                           <pattern>
<tag>              ::= symbol
<pattern>          ::= (symbol [<variable>]*)
<temporal>         ::= (AFTER <tag> <est> <lft>)
<est>              ::= integer
<lft>              ::= integer
<preference>       ::= ([symbol]*)
<variable>         ::= ?<name>
```

A procedure is defined according to its net–spec and its type. The basic syntax for the network net–spec is:

(tag pattern temporal–constraints preference–constraints)

where *tag* uniquely identifies a sub–task of the task, *pattern* is a reference which can be expanded by another DDO, *temporal–constraints* constrain the execution of sub–tasks relative to one another or the beginning of the whole group of tasks, and *preference–*

constraints constrain which DDOs can be used to expand the *pattern*. The basic syntax for the primitive net–spec is:

(*symbol variables*)

where *symbol* uniquely identifies an atomic action that can be carried out in the environment, and *variables* defines the information that is required to conduct the atomic action. Zero or more variables can be specified for a particular task. For example, in Figure 3.5, the taxi task requires information regarding which resource is to taxi as shown in the network field of the taxi-1 primitive. This is provided in the variable ?res.

```
(make-instance 'schema-data
     :name 'taxi
     :expands '(taxi ?res)
     :procedures
      (list
        '(TAXI-1 primitive
           (context (or (res-status '?res 'available)
                        (res-status '?res 'boarded)))
           (network (taxi ?res))))
...)
```

Figure 3.5: Portion of the Taxi Domain–Data Object from Pacifica

Effects

DDO effects specify what facts are expected to be present in the environment upon successful completion of the task represented by the DDO. This property allows the task schema object created from the DDO to have characteristics of a classical planning operator. Its purpose is to allow the REA to reason about the effects of DDOs to determine which task schemas could be executed to bring about specific effects in the environment. The present design does not incorporate such a mechanism though the provision of this information is for that purpose.

The general form of the effects slot is:

:effects '([<clause>]*)

```
<clause>    ::= (property entity value) |
                (property entity <variable>) |
                (property <variable> value) |
                (property <variable> <variable>) |
<variable>  ::= ?<name>
```

Uses–Resources

When the REA is to execute a task–directive it should be able to make reservations of resource utilization for each task of the directive by considering the uses–resources information from the Domain Data Objects. The REA would then be able to block out a temporal window for each resource required by a particular task according to its expected behavior. At present the resource reasoning capabilities of the REA are not sophisticated enough to use this information, but it is provided to assist in this respect.

The general form of the uses–resources slot is:

```
:uses-resources '([<variable>]*)

<variable>  ::= ?<name>
```

Exec-Cost

Execution cost information is provided in the DDO to allow the REA to reduce execution cost when multiple DDOs exist for carrying out a task. For example, we may have a situation where we need to move people from one location to another and there are two methods in which this could be done—by helicopter or by ground transport. By providing execution cost information, the REA could reason that if enough time existed then choosing the DDO with the ground transport would be best, but if time was critical then the cost of using the DDO with the helicopter would be best to achieve the task. Presently, it is assumed that the planning agent performs such reasoning before a plan is communicated to the REA for execution. However, these types of decisions are best made at execution time so this information is included in the DDO in the exec–cost slot. See Section 4.4.3 for a further discussion on this topic.

End-Cond

The end–cond of a DDO defines the circumstances in which the task represented by the DDO has successfully been achieved. The procedures of the DDO are executed until either the end–cond is satisfied or all procedures have been tried. When all procedures of a schema object have been tried and the end–cond remains unsatisfied the task fails.

The general form of the end–cond slot is:

:end-cond '<statement>

<statement>	::= (<L-connective> [<stmt>]$^{+}$) |
	(**not** <statement>) |
	<clause>
<stmt>	::= <clause> | <statement>
<L-connective>	::= and | or
<N-connective>	::= + | - | * | / | =
<R-connective>	::= <= | >= | eq | equal
<clause>	::= <ground> |
	(<N-connective> <expr> <expr>) |
	(<R-connective> <expr> <expr>)
<expr>	::= <clause> | <variable>
<ground>	::= (property entity value) |
	(property entity <variable>) |
	(property <variable> value) |
	(property <variable> <variable>) |
	value
<variable>	::= ?<name>

A clause may contain uninstantiated variables which become temporarily bound to test the validity of the clause. These bindings remain for testing the remaining conditions and may be used as values in subsequent condition clauses. However, these bindings are lost once the conditions have been tested. An example end–cond is:

```
:end-cond '(and (at '?res '?to-loc)
                (parked-at-gate '?res 'yes))
```

which states that the resources indicated by the variable *?res* be at the location indicated by *?to-loc* and that it be parked at the gate.

Duration

The temporal duration which a task can be expected to require to achieve its defined behavior is specified in the duration slot of its DDO. The duration is specified as:

:duration '([<time-spec>]$^{zero-or-one}$)

<time-spec>	::= (<eft> <lft>) | <time-function>
<eft>	::= numerical value
<lft>	::= numerical value
<time-function>	::= a function which returns (<eft> <lft>)

where *eft* is earliest finish time and *lft* is latest finish time. The specification of the duration slot is optional.

3.2.3 Task Schema Object

A Task Schema Object (TSO), or simply schema object, is created from network-node information in an IACL *synthesize* message (Section 5.2.7) and a DDO specification. It contains all slot definitions of the DDO along with additional information. The additional information contained in the schema object includes binding information, an instantiated pattern, a planner reference, pre- and post-node ordering constraints, execution status, owner information, and temporal information.

Each node in the network of the IACL *synthesize* message becomes a schema object during the synthesis process. From the planning agent's point of view each node in the network of the *synthesize* message is a primitive task which can be carried out in the environment. This may or may not be the case for the REA since the planning agent's knowledge may not require further decomposition in order to develop a plan. However, the REA may have to decompose a high level task into sub–tasks in order to reason about various aspects related to the task in order to guarantee successful completion in the environment. For this reason, the schema object represents a "classical" operator in the generative planning sense. The planning agent therefore has the ability to reason about tasks in the network without having to comprehend the actual complexities involved in executing the task.

```
#<Schema #XE4BD86>
    is an instance of the class SCHEMA:
 The following slots have allocation :INSTANCE:
 NAME                        NIL
 BIND-LIST                   NIL
 PATTERN                     NIL
 PLANNER-REF                 NIL
 PRE-NODES                   NIL
 POST-NODES                  NIL
 E-STATUS                    NIL
 T-DIRECTIVE-OBJ             NIL
 PROCEDURES                  NIL
 START-TIME                  NIL
 PROC-TRYING                 NIL
 PROCS-FAILED                NIL
```

Figure 3.6: Uninstantiated Task Schema Object

The task–schema object thus contains information to allow the REA to reason about a task (i.e., conditions, effects, uses-resources, duration, start-time, and ordering information). The task–schema object also containing information (see Figure 3.6) related to the execution of the task or sub–tasks which make up the high level task (i.e., procedures, repeat information, model analysis functions, and end–conditions).

As was stated earlier, the schema object is synthesized from information contained in the node–network of the IACL *synthesize* message as well as from a corresponding DDO definition. This section discusses those aspects of the task schema object that are not directly derived from the DDO definition.

Name and Bind–List

The name provides a unique identifier for the Task Schema Object.

The bind–list slot of the task schema object maintains the bindings made of the variables in the DDO from information provided in the pattern of a node in the node–network. The binding information is used when considering new or alternative sub–tasks to simplify the unification process. The general form of the bind–list slot is:

:bind-list ([(*variable* . *binding*)]*)

where *variable* is a input variable of the DDO matching the pattern from the node–network information, and *binding* is the value that the variable has been instantiated with through the pattern matching process.

For example, given node–network information of:

(node-3 (fly-transport c5-1 delta city-K) () ())

the fly-transport DDO would be selected. The input variables of its expands information would then be unified:

:expands '(fly-transport ?res ?to-loc ?from-loc)

and the bind–list slot of the task schema would contain:

((?from-loc . city-k) (?to-loc . delta) (?res . c5-1))

Pattern

The Pattern slot simply contains the pattern information given in the node–network. It is maintained explicitly in the task schema object as the specification of the schema (i.e., its purpose), and when communicating with the planning agent. Any node–network information not explicitly represented in objects with the REA is lost after the synthesis process.

Planner–Ref

The Planner–Ref information provides the REA with information regarding the "language" which the planning agent understands. That is, as schema objects are synthesized, they are assigned numerical references which are used internally in the REA for processing purposes. If the REA were to communicate information to the planning agent using its internal referencing scheme the planning agent would not understand

what the REA was referring to. Therefore, the REA maintains the planning agent's reference to the task so that while communicating they will both have a common point of reference.

Pre–Nodes and Post–Nodes

The Pre–Nodes and Post–Nodes slots store the ordering constraint information that is directly related to a specific node in the node–network. The Pre–Nodes information states the high level task(s) which must have been completed before a specific task can begin execution. The Post–Nodes information provides guidance to which task(s) are to follow the successful execution of a high level task.

E–Status

The execution status, or E–status, of a task schema object can be instantiated with one of four values at any give time: **U**nscheduled, **R**eady-for-processing, **P**rocessing, or **F**ailed. These values inform the control mechanism of the REA of the state of execution of the task represented by the schema object, and allow for appropriate processing to be conducted.

T–Directive–Obj

The T–Directive–Obj slot provides a reference to "plan" to which the task schema object belongs. Its primary purpose is to provide information to the control mechanism of the REA so the controller can decide which task to execute next according to priority.

Procedures

Each procedure of a DDO (refer to Section 3.2.2) that makes up a particular task is synthesized into a procedure object (see Figure 3.7). The procedure objects are what the control mechanism manipulates to cause the REA to dispatch and execute tasks in the environment.

```
#<PROCEDURE LOAD-2>
    is an instance of the class PROCEDURE:
 The following slots have allocation :INSTANCE:
 NAME              LOAD-2
 ITYPE             LOAD
 E-STATUS          P
 OWNER             #<TASK-DIRECTIVE T-DIRECTIVE-1>
 PARENT            #<PROCEDURE FLY-TRANSPORT-1>
 SERVICING         #<SCHEMA NODE-1.0 FLY-TRANSPORT>
 P-TYPE            NETWORK
 EXEC-PROCS        (#<PROCEDURE LOAD-CARGO-PLANE-1>)
 SUCCESS           (OR (LOAD-STATUS 'C5-1 'LOADED)
                       (AND (NATIONALS-AT 'CITY-K 'NO)
                            (NOT (O-TYPE 'CITY-K 'BASE))
                            (CARGO 'CITY-K 'NO)))
 EXECUTED          (PROCNET-40 PROCNET-39 PROCNET-38)
 FAILED-P          NIL
 START-TIME        943
 TEMPORAL-INFO     ((PROCNET-40 1232)
                    (PROCNET-39 1231)
                    (PROCNET-38 1229))
 CONTEXT           (AND (O-TYPE 'C5-1 'AIR-CARGO-TRANSPORT)
                        (AT 'C5-1 'CITY-K))
 ORDERING          ((PROCNET-38 NIL (PROCNET-41))
                    (PROCNET-39 NIL (PROCNET-41))
                    (PROCNET-40 NIL (PROCNET-41))
                    (PROCNET-41 (PROCNET-38 PROCNET-39 PROCNET-40) NIL))
 NETWORK           (#<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
```

Figure 3.7: Load-2 Procedure Object from the Load DDO

The Procedures slot of the task schema object maintains a list of the procedure objects which make up the high level task represented by the schema object. However, only the first decomposition of procedure objects are held by the high level schema object. For example, the high level task of fly-transport would have a corresponding schema object whose Procedures slot would contain:

```
(#<PROCEDURE FLY-TRANSPORT-2> #<PROCEDURE FLY-TRANSPORT-1>)
```

where each of these procedures must be further decomposed into their constituent procedure objects for each of their sub–tasks.

Start–Time

The Start–Time slot information is used when temporal constraints exist for sub–tasks of a high level task. The slot is instantiated with a timestamp when the task or the first sub–task is dispatched to the environment for execution.

Procs-Trying and Procs-Failed

The Procs-Trying and Procs-Failed slots store information related to the procedures which are being used to achieve the task and those procedure which have been tried that have failed. This information assists the control mechanism in determining the means which have been tried to bring about the desired effects of the task.

Other Information

The Task Schema Object (TSO) additionally contains information from the DDO that specified the execution time behavior of the task which the TSO represents. This information includes conditions, repeat–while, calc, effects, uses–resources, end-cond, and duration.

3.2.4 Monitor Object

Monitor Objects are used to assist the REA in monitoring the execution of the Task Directive by informing it about what to expect to be true in the environment throughout the execution of the directive. This information is provided relative to the high level tasks contained in the node–network information.

For each Causal Structure Record (CSTR) in an IACL *synthesize* message (Section 5.2.7), a monitor object is synthesized that is responsible for notifying the REA if values of monitored entities are not as expected over specified ranges of execution (Figure 3.8). A Monitor object is only "active" during the execution of the Task Directive to which it belongs, and during the interval defined by the range–start to range–end. Chapter 7 will describe what is meant by causal structure and discuss how monitors are synthesized and used to detect potential execution failures. Here we will examine the

```
#<MONITOR-7 (AT GT1) T-DIRECTIVE-1>
    is an instance of the class MONITOR:
The following slots have allocation :INSTANCE:
NAME                    MONITOR-7
TAG                     CSTR-1
RIGIDITY                :F
TASK-D                  #<TASK-DIRECTIVE T-DIRECTIVE-1>
SCHEMA                  #<SCHEMA NODE-10.0 DRIVE>
EXPECTED-VALUE          DELTA
KNOWN-CONTRIBUTORS      (1.0)
BEING-MONITORED         (AT GT1)
RANGE-START             1.0
RANGE-END               10.0
```

Figure 3.8: Monitor Object for (at gt1)

information contained in the slots of the Monitor object.

Tag

The Tag provides a means to inform the planning agent exactly where in the execution of a plan a failure has occurred. When a Monitor object detects a discrepancy between the value of what it is monitoring has and what it is expected to have during its protection interval, a violation is said to have occurred. When a violation occurs, the REA may notify the planning agent, and in doing so it uses the Tag information as a common point of reference when communicating.

Rigidity

The Rigidity information relates to how hard or soft a violation of a causal structure range should be taken. That is, it provides information to tell the REA, in the case where it detects values that are other than those expected, whether the violation should be treated as a failure (i.e., hard constraint) or whether the REA should try to reestablish the value expected (i.e., flexible constraint). This rigidity information is derived from the condition types used in O–Plan [Drabble *et al.* 94], and was originally motivated by McDermott's work on transformational planning [McDermott 92].

Task-D and Schema

The Task–D slot is a reference to the Task Directive to which the Monitor object belongs. The Task–D slot contains a reference to the Task Directive so that the Monitor object is sure to be active only when its associated Task Directive is being executed by the REA.

The Schema slot provides a reference to the Task Schema Object where what the Monitor object is monitoring should have the value expected.

Expected–Value

The Expected–Value slot maintains the value of what is being monitored is expected to be during the protection interval. That is, the interval defined by Range–Start to Range–End.

Known–Contributors

The Known-Contributors slot maintains a list of nodes from the node–network information of the *synthesize* message whose successful execution yields the expected value. This list only contains nodes that are known to be temporally or sequentially constrained to be executed before the expected value is required (i.e., at the node specified by Range–End).

Being–Monitored

The Being–Monitored slot specifies what the Monitor object is monitoring. The property of an entity in the environment that is to have some expected value is maintained in the Being–Monitored slot. For example, when the Monitor object (shown in Figure 3.8) is active it checks the World Model to make sure that the *at* property of the gt1 resource has the value of delta in the range of Node-1.0 to Node-10.0 during the execution of Task-Directive-1.

Range-Start and Range-End

The protection interval, or the interval during which what is being monitored is to have the expected value is defined by the slots Range–Start and Range–End. Range–Start defines the node which is the primary contributor of the expected value, and Range–End defines the node which is expecting what is being monitored to have the expected value.

3.2.5 Behavior Objects

Humans use two basic types of behaviors to keep abreast of changes in the environment in which they are situated—periodic and contextual. Periodic behaviors allow us to repeatedly initiate a behavior on a given temporal frequency. For example, while driving a car we need to check our rear view mirror and instruments every 10 seconds. Contextual behaviors allow us to initiate a behavior when a particular stimuli is present. For example, when going to an automated teller machine we need to flee if someone attempts to rob us. These two types of behaviors are innate. That is, everyone of us possesses the ability to exhibit behavior periodically and contextually.

```
#<Active-Behavior #XE4B67E>
    is an instance of the class ACTIVE-BEHAVIOR:
 The following slots have allocation :INSTANCE:
 NAME          NIL
 PROPERTY      NIL
 FREQUENCY     0
 BEHAVIOR      NIL
 EVENT         NIL

#<Passive-Behavior #XE4B07E>
    is an instance of the class PASSIVE-BEHAVIOR:
 The following slots have allocation :INSTANCE:
 NAME        NIL
 HANDLES     NIL
 TRIGGER     NIL
 ACTION      NIL
```

Figure 3.9: Uninstantiated Active and Passive Behavior Objects

For the REA to be considered intelligent by the types of behavior it exhibits, it too must possess an innate behavior mechanism. The active and passive behavior constructs of the TBL allow the REA to exhibit these types of behavior (Figure 3.9). Active behavior objects allow the REA to exhibit periodic behavior, and passive behavior objects allow for context–based behavior to be exhibited. The use of these behavior types will be discussed further in Section 7.3, however in this section we will discuss the information contained in the objects defining the behaviors.

Active Behavior

The active behavior is defined by five attributes: name, property, frequency, behavior, and event. The name attribute provides a unique identifier for the behavior. The property attribute provides information necessary for accomplishing the behavior when it is activated. For example, if the behavior was to issue a sensor request then the property attribute could be used to select a particular entity to be sensed. The frequency attribute defines the period of activation for the behavior. Specified in seconds, it is used by the Active Behavior Manager to determine the next time the behavior should be activated. The behavior to be exhibited upon activation is defined by the behavior attribute. This information is used by the capability specified in the event attribute to determine what should be done once the behavior has become activated. Finally, the event attribute defines the capability that is responsible for processing the behavior once it has become activated.

Passive Behavior

The passive behavior is defined by the four attributes: name, handles, trigger, and action. The name attribute provides a unique identifier for the behavior. The handles attribute defines the situations that the behavior is intended to be used for. For example, if we define a behavior that is to be activated when fires are detected then the handles slot may contain oil-fires and/or gas-fires specifying that if oil or gas fires are detected then this behavior it appropriate. The trigger attribute specifies the conditions under which the behavior is to become activated. Finally, the action

attribute defines the capability that is responsible for processing the behavior upon its activation. When an passive behavior is installed, two things happen: first, the information in the handles attribute is used to identify the particular situations in which the capability specified in the action attribute is appropriate for. This assists the REA when failures are detected by identifying specialist knowledge that is available for specific situations (see Section 7.4 for more detail); second, the behavior is placed on the untriggered agenda until which time the conditions specified in the trigger attribute become satisfied.

3.3 Chapter Summary

This chapter has presented the World Model of the REA, and the object–oriented, hierarchical, Task Behavior Language.

The World Model assimilates information from the environment which is then used by the REA as a basis for decisions at runtime. We have seen the design of the model, the query mechanism, and how it addresses persistent facts.

The TBL is modeled on Firby's RAP approach and adds additional constructs to that language to provide the REA with more information to better make decisions at runtime.

The next chapter will introduce the design and architecture of the REA as well as explaining how the constructs of the TBL are used during execution.

Chapter 4

REA Architecture and Control

To this point we have considered a set of characteristics necessary for providing rational behavior, and a language to describe the behavior of tasks. We still require a means which will allow these to be integrated to understand how together they help to provide the behavior we desire in a dynamic environment. Thus, we still need to discuss the underlying architecture of the REA and its control mechanism.

The early designs of planning/execution systems separated the responsibilities of deliberation and execution (*cf.* [Fikes *et al.* 72a,Wilkins 84,Dean 84]). In that approach the deliberation process was responsible for generating a provably correct plan of action which was then "thrown over the wall" to an awaiting execution system that monitored the execution of that plan. However, this approach only met with limited success. The reasons for this were threefold. First, because the world has the potential to change while planning is in progress causing substantial replanning to take place with little execution occurring. Second, with the uncertain effects of actions, seemingly correct plans may fail to achieve their goals [Chapman 87]. Third, the knowledge required by the planning agent to generate a plan and that of the execution agent to execute the plan are often at different levels of abstraction leading to problems of representation and tractability.

These dilemmas led researchers to develop so called *reactive planners*, which were not planners at all. They were attempting to design systems that had an appropriate action for every possible situation and a sensing system that was able to identify all aspects of the world so a mapping could take place to execute appropriate actions in

the correct situations [Ginsberg 89].

Once again there were problems with the approach. Firstly, designers of these systems had to address each possible situation the system might find itself (which is unrealistic for all but trivial domains). Secondly, execution–time decisions tend to be myopic and thus the robustness of these systems were limited [Lyons & Hendriks 92]. This is not to say that the reactive planning approach has not had its successes [Brooks 86,Agre & Chapman 87,Kaelbling 88,Connell 92], but the problem was their lack of flexibility and the effort required to utilize them in new domains.

The limitations of the reactive planning approach led others to attempt to integrate planning systems with execution systems [Wilkins 85a,Ambros-Ingerson & Steel 88]. However, this integration identified the problem with balancing deliberation and reaction [Hendler 90]. The problem is to decide, for a particular situation, whether it is better to spend more time deliberating to develop a solution (with greater utility in addressing global concerns such as resource utilization or minimum number of operations) at the expense of possibly lost opportunities. Or is it better to react and take an opportunity that is present at the expense of global optimization or causing additional interactions (since only local concerns are considered when reacting).

This deliberation/reaction problem typically manifests itself in systems where a single agent is responsible for both deliberation and reactivity. In such systems it is an important issue. However, the current trend in the AI planning and robotics communities is to separate these responsibilities in two asynchronously operating agents (*cf.* [Firby 89,Lyons & Arbib 89,Bonasso & Slack 92]). The difference from the original approach where deliberation and execution responsibilities were separated is that now the deliberation system attempts to adapt the behavior of the execution system and the execution system possess more intelligence to deal with execution failures and the ability to take advantage of opportunistic situations that will allow for the achievement of its goals.

The design of the REA architecture and control mechanism follows this latest trend in execution agent design. The architecture is divided into five independent components each with its own responsibilities and expertise. Together they form an agenda–based, asynchronous computational facility which provides the means to integrate the basic

set of characteristics and the necessary functionality to behave rationally.

Built on top of this architecture is the control mechanism which orchestrates the behavior exhibited by the REA. The controller's job is to prioritize intentions held by the REA and to apply various sources of knowledge to bring about the desired behavior.

In this chapter we will discuss the design of the REA. This includes the design goals, design motivation and a characterization of the design. This will be followed by a discussion of the REA Architecture and the components which make up that architecture. Finally, we will look at task execution and control using this architecture considering a detailed example and discussing how control decisions regarding the selection of alternatives is made.

4.1 REA Design

To this point the primary emphasis has been on examining previous rational agent designs for similar features in order to characterize those capabilities which define rational behavior (Chapter 2). The problem remains, of course, how we integrate these capabilities into a single design which yields a robust rational REA.

One such design is presented here[1]. This design brings together previous approaches to rational agent design in order to draw upon the methods used for the implementation of the basic REA characteristics. In addition to providing full support for all of the basic characteristics, the design also provides mechanisms for reasoning about the knowledge, capabilities and commitment of a superior agent, and a communication protocol specification (Chapter 5). The communication protocol allows the REA to acquire new knowledge, additional capabilities, and receive assistance when circumstances necessitate.

The goal is to develop a system which exhibits the following characteristics.

- The ability to acquire new knowledge and capabilities. Using the Inter–Agent Communication Language (IACL) the agent is able to to acquire additional

[1] We are not stating that this is the only possible design. There could be other designs equally as good or better however, this design will be the one discussed in this research

knowledge from the planning agent in addition to, that normally acquired from the environment via sensors (see Section 5.2.7). The IACL also enables the agent to acquire new processing functionality in the environment from the planning agent (see Section 5.2.3).

- Possess a reasoning mechanism which allows the REA to reason about commitment of the planner to tasks, priority of tasks, knowledge and capabilities of the planner and itself, and about goal and fact driven entities.

- The ability to accept plans which constrain or partially constrain action selection. The REA will tend to be myopic when having to make a decision at execution time, yet it should choose the most appropriate action for the immediate circumstances. The planner on the other hand has the ability to take more global constraints into consideration. Therefore, the agent should be able to constrain its selection when directed by the planner and be free to make what it feels is the best choice when the planner does not provide any constraints.

- Be independent and functionally separated from the planning agent yet loosely coupled to be able to receive assistance when required.

- Have the ability to recover from failures either through replanning tactics or assistance from the planning agent.

- Possess all of the REA capabilities—guaranteed response, failure recovery, innate behavior, asynchronous events, weighing alternatives, change of focus, predictability, and temporal reasoning as defined in Chapter 2.

4.2 Design Motivation

The motivation behind the design of the control mechanism, and subsequently the architecture, was not to develop a conceptually new approach, but rather enhance an existing approach. Firby's RAP execution model [Firby 89] was the most likely candidate as the basis for the REA design for two reasons. First, the robot intelligence community has begun to agree on a generalized robot software architecture whose middle "sequencing" layer was envisioned by Firby [Bonasso & Slack 92]. Second, the

RAP approach has influenced a multitude of execution and integrated architectures, so its validity is strongly supported. These include:

- FLOABN [Alterman *et al.* 91]
- TOA [Lewis 91]
- RAP/DMAP [Martin & Firby 91]
- CASTLE [Collins *et al.* 91]
- ATLANTIS [Gat 91]
- Meliora [Whitehead 91]
- RAP/GAPPS [Slack 92]
- XFRM [McDermott 92]
- 3T [Bonasso & Barratt 93,Elsaesser & Slack 94]
- PARETO [Pryor 94]

Thus the primary design goal for the REA architecture and control mechanism was to use Firby's RAP approach as a foundation, and to augment it to allow for:

- Taskability
- Modularity
- Flexibility
- Concurrent Task Execution
- Object-Oriented Control Knowledge

4.2.1 Taskability

To say that an intelligent agent *must* be taskable is a far cry from the truth. However, any intelligent agent that is to interact with others, whether they be human or otherwise, must be taskable. The domain of space flight has been the driving vision for the

taskability requirements of the REA: not that any other domain where the agent is autonomous would require less taskability, but with the agent being a satellite or rover you lose the ability to add or change aspects of the agent once it has left the launch pad.

Taskability for the REA is in the form of Inter-Agent Communication Language (IACL) messages. With the IACL message protocol (Chapter 5) the REA is able to accept tasks, reject inappropriate tasks, report the status of tasks, and abort task execution.

4.2.2 Modularity

There are distinct roles to be played in an execution agent: communication, intention selection, control, world model assimilation, etc. Therefore, to study the influence of each of those roles we chose to modularize the architecture along those lines using the general O–Plan architecture as a base [Tate *et al.* 92]. This provides the ability to replace components, thereby altering functionality and allowing us to explore more complex roles or more efficient ways of performing the same roles. If agents such as the REA are to come into mainstream use, we are going to need to build them in such a way as they can be maintained [Minton 91].

4.2.3 Flexibility

Like the requirements for taskability, the domain of space flight has been a primary driving vision for flexibility. Flexibility for the REA comes in the form of knowledge, capabilities, and active/passive behaviors, and the fact that each can be adapted via the IACL message protocol. Knowledge can be added to allow the REA to carry out new tasks or provide new means of achieving tasks. Knowledge can also be removed to prevent the REA from attempting undesired tasks. Capabilities can be added (or removed) to alter the processing behavior, and active/passive behaviors can be added or removed to adapt the REA's innate behavior. See Chapters 5 and 7 for further discussion.

4.2.4 Concurrent Task Execution

One of the shortcomings of the RAP Approach [Firby 89] was that of unrealistic primitive actions. That is, he assumed that primitive actions could only be executed one at a time. No further task expansions could be considered until results from a primitive action were returned by the hardware interface. This limitation was overcome in the work described here by providing ordering constraints information which allowed for concurrent execution (see Section 3.2.2), and employing control algorithms to exploit that information (see Section 4.4.2).

4.2.5 Object-Oriented Control Knowledge

Each entity which the REA must reason with or about is represented as an object. That is, Domain Data Objects are used to represent tasks and the procedures by which tasks can be dispatched to bring about effects in the environment. In addition to representing a task itself, the approach allows the objects to maintain other information directly relevant to the task concisely and efficiently. This information includes execution status, processed procedures, processing procedures, procedures which failed, and temporal information. This approach eliminates the need to store (and retrieve) such information in the World Model. Thus, execution decisions are taken more efficiently by having the objects directly represent information necessary to take control decisions.

This approach would be advantageous in a cooperative environment. For example, if one agent were becoming overwhelmed with a large task load it could offload tasks to other agents. Using the approach presented here for Task Directives it would be a simple matter of passing the object representing the task to be offloaded since it contains all information related to the task.

4.3 The REA Architecture

To achieve the integration of the capabilities required by the design (Section 4.1) the underlying architecture must provide certain functionality. That is, it must be able to

maintain multiple tasks with varying priorities. It must be able to easily change its processing behavior by adding or removing declarative knowledge of such behavior. It must possess a triggering mechanism to select appropriate behaviors based upon events and temporal deadlines, and must be able to accept asynchronous inputs.

The REA architecture[2] is composed of five components: *Communication Manager, Agenda Manager, Knowledge Platform, Trigger Manager*, and *Active Behavior Manager*. The following sections describe the role and function of each component of the architecture in detail. This is followed by a discussion of the flow of control through the architecture.

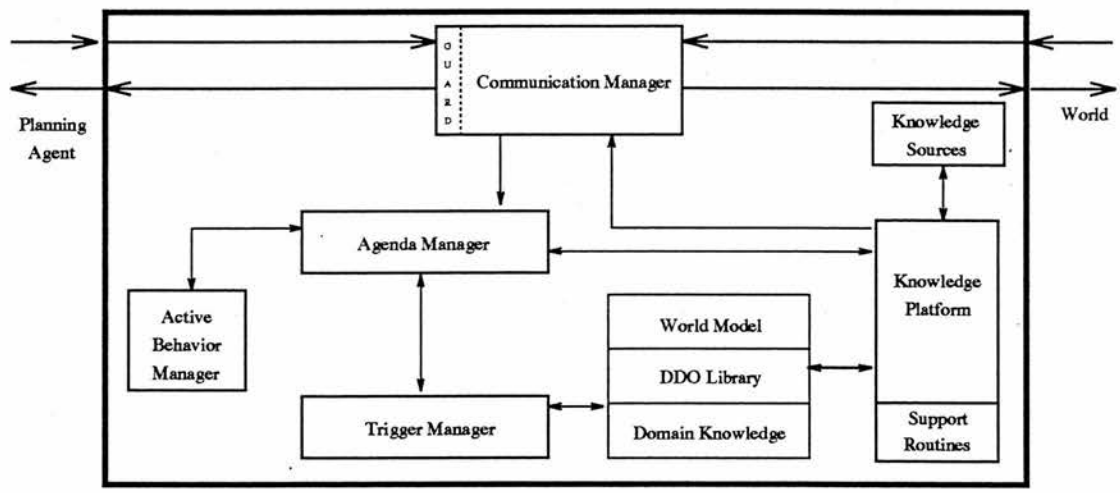


Figure 4.1: REA Architecture

4.3.1 Information Flow and Architecture Summary

The aim of this section is to describe the flow of a single typical cycle through the architecture, see Figure 4.1. The main steps are as follows:

[2] The agenda–based reasoning architecture of O–Plan [Tate *et al.* 94] was chosen as the basis of the architecture for the REA as it provided some of the basic means to meet the requirements of the design. This architecture was adapted to fully meet the requirements of the design by adding functionality not already present, such as automatic initialization, implementation of the Guard concept, control information in agenda records, the ability to trigger waiting untriggered agenda entries, and the ability to delete agenda entries.

1. An event is received by the Communication Manager (Section 4.3.2) and checked against the current knowledge and capabilities of the REA. If the event is valid then the event is converted to an intention (i.e., agenda entry) and passed to the Agenda Manager (Section 4.3.3).

2. The intention is immediately passed to the Trigger Manager (Section 4.3.5) to await triggering according to conditions of applicability for the intention.

3. Once the intention becomes triggered it is sent to the Agenda Manager to await its turn to run on the Knowledge Platform (Section 4.3.4) where it will be processed by a capability (i.e., knowledge source) appropriate for the particular intention.

4. On completion of processing by a capability the Knowledge Platform signals the Agenda Manager that it is idle and a new agenda entry should be sent. The Agenda Manager then sends to the Knowledge Platform the intention from the top of its triggered list.

5. On receipt of the intention the Knowledge Platform retrieves the code body of the specified capability from its local library (if not already loaded) and processing of the intention begins.

6. The capability will either run to completion or signal a failure. In either case the Knowledge Platform returns to the Agenda Manager any new intentions which have been generated by the capability.

7. Any intentions passed to the Agenda Manager are processed as in step 2.

With each of the major processes running *asynchronously*, new event entries can arrive, new tasks can be set by the planning agent, etc., as agenda entries are being processed within this cycle.

Agenda entries are posted onto the agenda which is simply a list of outstanding intentions to be performed, and are selected by the AM to be processed according to the REA's capabilities. The knowledge sources provided represent the knowledge of the system and are referred to as *capabilities*. Knowledge sources run on the Knowledge Platform which is able to run all of the available knowledge sources, albeit one at a time.

The Trigger Manager and the Active Behavior Manager (Section 4.3.6) are responsible for determining when intentions held by the REA are contextually or temporally valid on each cycle, respectively. When an intention is valid it is said to be *active* and is posted to the AM for management.

The REA is given tasks by adding intentions to its agenda. The AM is responsible for selecting an outstanding intention to process whenever a knowledge source can be activated on the waiting Knowledge Platform.

Domain information, information about the tasks which can be carried out in the environment, and the REA's model of the environment in which it is situated are consulted by capabilities (i.e., knowledge sources) as they run. Thus, the capabilities have access to task descriptions (which define higher level activities in terms of more detailed activities), as well as, definitions of resources, other domain constraints, and information about its environment.

4.3.2 Communication Manager

The Communication Manager (CM) provides an end point for *inter-agent* communication. It is responsible for all communication into and out of the REA. The REA has the ability to communicate both **left** and **right**. The agent to the **left** is the superior agent and to the **right** is the subordinate agent. Communication to and from the left is conducted according to the IACL protocol (Chapter 5). To the right, communication is conducted in a form suitable to the agent, hardware, or simulator to which the REA is connected.

There are four communication channels —*leftin*, *leftout*, *rightin*, and *rightout*— on which information either arrives or is sent via the CM. When information arrives on either the leftin or rightin channel it is termed an *event* and the CM is informed of the new information by means of an *event entry*.

The basic job of the CM is to convert the event entry into an agenda entry which is then passed to the Agenda Manager in order to add it to the list of intentions to be processed. However, as information arrives on the leftin channel, additional processing takes place by the leftin Guard of the CM before the event entry is passed along.

The role of the leftin Guard is to accept or reject information received from a superior agent to the left. Its responsibility is twofold. First, it determines whether the message itself will be understood by the REA. The Guard uses the current capabilities of the REA as the criteria for making such decisions. If the message is of a type such that the REA would not be able to process the message then the message, and subsequently all the information which it contains, is rejected with an *unknown-capability* message of the IACL Protocol (Section 5.2.8) to the originator of the message. The capabilities of the REA are composed of the knowledge sources currently in its library. The second responsibility of the leftin Guard is to determine whether the REA is able to understand the contents of the message. The Guard uses the capabilities of the REA as the basis for making such decisions. If the REA will understand how to process the message, but does not possess the knowledge to carry out one or more tasks specified in the message then it is rejected. This is done via an *unknown-knowledge* message of the IACL Protocol (Section 5.2.9) .

The underlying concept of the Guard is to mimic the functionality of the human ear. When someone speaks to us in French and we do not understand French then we are unable to process any information that they may be trying to convey. The same applies when someone speaks to us in a language we understand and we are unable to do what they ask because they either have used terminology that we do not understand, or have asked us to do something that we do not know how to perform. There is no purpose in having the REA attempt to achieve tasks which it has no hope of satisfying due to lack of knowledge or understanding. We want a faster short–circuit response to event communication. The Guard fills this niche in the REA.

4.3.3 Agenda Manager

The Agenda Manager (AM) manages the intentions of the REA. It is responsible for ordering the active intentions (according to a prescribed ordering function), and accepting additional intentions from knowledge sources. Intentions in the REA architecture nomenclature are called agenda entries (AE).

Agenda entries have the form:

```
(<agenda-tag> <priority> <trigger> <ks-name> <body> <controller-info>)
```

where *agenda-tag* uniquely identifies the intention, and *priority* specifies the processing priority or urgency with which the AE should be processed. The *trigger* specifies the conditions that must be satisfied or the context in which the AE may become active (i.e., triggered and available for processing). The *ks-name* information informs the Knowledge Platform of what capability (i.e., knowledge source) should be used to

```
<AE-82/0:
     PRIORITY: 95
     TRIGGER: (:WAIT-ON-EFFECT-GROUP
               ((AT 'REA::C5-1 'DELTA)
                (PARKED-AT-GATE 'C5-1 'YES)))
     BODY: (EXECUTE <PROCEDURE FLY-TRANSPORT-1>)
     CONTROLLER-INFO: (C5-1)>
```

Figure 4.2: Formatted Agenda Entry

process the information contained in the AE, and the *body* contains the arguments or local information which the knowledge source assigned to process the AE will require when it is called to process the intention. Finally, the *controller-info* field provides information necessary for the controller to carry out operations on the AE, such as determining the Task Directive the AE belongs to or determining the resources required to execute a particular task (see Figure 4.2).

The AM accepts agenda entries from three sources:

1. new agenda entries from the Communication Manager,
2. triggered agenda entries from the Trigger Manager, and
3. new agenda entries from a processing capability.

The AM is also responsible for assigning a priority to the agenda entry before it is sent to the Trigger Manager to await triggering. Agenda entries are prioritized using numerical values that are specified according to the capability necessary to process

Knowledge Source	Priority
INIT	1000
RESOURCE	251
DISPATCH	250
AGENDA	200
DIRECTIVE-FAILURE	200
SENSOR	200
WORLD	200
FAILURE	195
SYNTHESIZE	150
ADD-BEHAVIOR	100
ADD-CAPABILITY	100
ADD-KNOWLEDGE	100
SUPERVISE	90
EXECUTE	80
DEFAULT	10

Table 4.1: REA Capability Priorities

the AE[3]. In order for the REA to have the ability to change its focus of attention it needs some mechanism to determine the importance of the various intentions it holds. The priority mechanism of the AM serves this purpose allowing it to choose the most important intention to process next. Table 4.1 shows the fundamental REA capabilities and their static priorities[4]. These priorities are arbitrary as to their numerical significance, but are critical in how they dictate processing order of certain capabilities (i.e., the ones given in Table 4.1) with respect to one another. The default priority is used as the priority for all non-explicitly named capabilities: those which are not directly involved in the *control* of the agent's behavior.

Agenda entries are held in two separate structures in the REA.

1. The triggered agenda entries (i.e., those entries which are ready to run on the knowledge platform) are maintained by the AM.

2. The untriggered agenda entries (i.e., those entries whose conditions or context

[3] The exception to this rule is that IACL *synthesize* messages specify additional priority for Task Directives which correspond to the planning agent's commitment to the directive. Therefore, the priority of an intention from a Task Directive is calculated by the sum of the priority of the necessary capability and the priority of the Task Directive.

[4] In the current implementation the priorities are given statically; however this need not be the case.

have yet to be satisfied) are stored in the Trigger Manager.

The AM manages the list of triggered agenda entries or intentions and upon request from the Knowledge Platform sends the agenda entry at the top of the triggered list for processing. The list is dynamically maintained with the intention at the top of the list being the next one to be sent to the Knowledge Platform. New agenda entries which are received from the CM are immediately sent to the Trigger Manager to await triggering. When the Knowledge Platform becomes idle the AM sends the intention from the top of the triggered list. The triggering of intentions is managed by the Trigger Manager (see Section 4.3.5).

4.3.4 Knowledge Platform

The Knowledge Platform (KP) is the component of the architecture responsible for performing the capabilities supported by the REA. It executes the code bodies of the knowledge sources contained in the REA's Knowledge Source Library and calls on support routines to interact with the other components of the architecture.

The KP has the ability to run a single capability at a time[5]. When the KP is free (i.e., available to process a capability) it signals the AM. The AM selects an intention to process and sends its corresponding Agenda Entry (AE) to the KP. The KP determines whether it has the capability required to process the AE presently loaded according to the ks-name information of the AE. If so, processing of the intention begins. Otherwise, the ks-name information is used as an index into the Knowledge Source Library to load the appropriate capability before processing begins[6].

The primary knowledge sources (i.e., capabilities) of the REA found in its Knowledge Source Library are as follows.

Active Behavior Performs the necessary tasks to execute an active behavior.

Add Behavior Converts information from IACL Add-Behavior messages to

[5] The general O-Plan architecture underlying that of the REA is designed to allow for multiple KP's however, this feature has not yet been exploited.

[6] Preloading of the capabilities is possible in the architecture to eliminate the time dynamic loading of capabilities requires.

	either active, passive, or specialist behaviors depending upon the type specified in the message.
Add Capability	Converts information from IACL Add-Capability messages to knowledge sources which can then be used by the REA.
Add Knowledge	Converts information from IACL Add-Knowledge messages to Domain Data Objects and stores them in the REA's internal data structures for later use by the REA.
Agenda	Performs specific functions on the triggered and untriggered agenda lists. The functions include the removal of specified agenda entries and the retriggering of agenda entries.
Dispatch	Sends messages to the hardware interface, simulator, or agent being controlled by the REA. The messages are converted to the appropriate type for the entity to which the REA is connected and then dispatched.
Execute	Executes procedure objects either by decomposing network procedures or by posting agenda entries to the AM for dispatching primitive procedures.
Failure	Handles specific types of anticipated failures from the environment in which the REA is situated. It first looks for specialist capabilities to address a failure. Failing that, it tries to apply some general knowledge to address the failure. If all else fails, it posts an AE to the AM to notify the planning agent that the failure is beyond its knowledge and/or capabilities.
Init	Initializes the REA. It establishes a monitor process to trigger monitors upon updates to the World Model, synchronizes its internal clock with that of the entities to its right, loads in the domain specific TBL, and acquires information regarding the REA's knowledge and capabilities to be used by the Guard on the leftin channel.

Notify Sends IACL messages to the planning agent via the CM's leftout channel. For example, IACL execution-failure, no-capability, and no-knowledge messages.

Remove Capability Deletes a specific capability held by the REA as specified in an IACL Remove-Capability message.

Remove Knowledge Removes a specific Domain Data Object from the REA's internal data structures so that it is no longer available for use by the REA. The specific DDO to be removed is specified in an IACL Remove-Knowledge message.

Resource Checks for resource conflicts before tasks are dispatched. It first determines whether the REA has already dispatched a task requiring the same resource as the task it is about to dispatch. If so then dispatching is delayed until the task that is using the resource completes. It also tries to determine if some other agent in the environment has been allocated the resource in question by issuing sensor requests about the resource.

Sensor Handles the sensor requests from active sensing. It dispatches appropriate sensor requests if the frequency constraints of the sensor allow for use of the sensor. Otherwise, the active behavior is set to trigger on the next period when the sensor can be used. The behaviors are not given back to the ABM if the monitor no longer exists (i.e., its protection range has expired).

Supervise Supervises the execution of Task Directives. It posts agenda entries to the AM for execution of each high-level task in the Task Directive according to ordering constraint information.

Synthesize Converts information from an IACL Synthesize Message into a Task Directive Object and creates Monitor Objects from Causal Structure Record (CSTR) information in the message.

World Updates the World Model according to information received on the rightin channel. It uses models of the various sensor types known to the REA to make updates when sensor request information is returned from the environment.

4.3.5 Trigger Manager

The Trigger Manager (TM) is responsible for maintaining the untriggered intentions of the REA and determining when intentions have become active (i.e., triggered). The trigger of each untriggered AE held by the TM is checked as new information is asserted into the World Model. As AE's become triggered they are passed to the Agenda Manager to await processing by the appropriate capability.

```
(:WAIT-ON-EFFECT-GROUP
        ((AT 'C5-1 'DELTA)
         (PARKED-AT-GATE 'C5-1 'YES)))

(:WAIT-ON-EFFECT
        (AND (>= (FUEL-LEVEL 'C5-1)
                 (* *AIR-THRESHOLD* (MAX-FUEL 'C5-1)))
             (AT 'C5-1 'DELTA)
             (LOAD-STATUS 'C5-1 'EMPTY)))
```

Figure 4.3: Conjunctive Triggers with Different Trigger Types

Presently, two types of triggers are used. These are *wait-on-effect-group* and *wait-on-effect*. The type of the trigger informs the TM how it is to interface to the World Model (WM) to determine if the trigger is satisfied. The *wait-on-effect-group* type specifies that the trigger is composed of two or more conditions that are to be treated as a conjunction. That is, in order for the trigger to be satisfied, all conditions must be satisfied in the World Model. The *wait-on-effect* type is more general and allows for any conjunctive, disjunctive, boolean comparison, or combination thereof to make up a query to the WM (see Figure 4.3).

4.3.6 Active Behavior Manager

The Active Behavior Manager (ABM) is responsible for maintaining a list of active behaviors which are awaiting activation at a particular point in time. The behaviors awaiting activation are held in ascending time order with the lowest value being the next behavior to be triggered. When a behavior is triggered (i.e., the timestamp matches the current clock value) the ABM passes the behavior to the AM so it can be processed by the appropriate capability.

A behavior is an innate ability possessed by the REA. As was discussed in Chapter 3, there are two types of behaviors—active and passive. The distinction being that active behaviors occur on a periodic basis and passive behaviors are opportunistic in nature and only become "active" when the conditions exist to trigger the behavior.

Active behaviors maintained by the ABM have specified frequencies for their occurrence. When an active behavior is given to the ABM it assigns a time stamp to the behavior and adds on the behavior's frequency to that value giving the next time the behavior is to occur. When that time arrives, the ABM sends the behavior to the Agenda Manager as an agenda entry so that it can be processed by the Active-Behavior capability of the REA. Once processed (i.e., the specific behavior has been carried out), the behavior is given back to the ABM so a new time stamp can be assigned and the behavior can occur again in the future according to its frequency of occurrence.

4.4 Task Execution and Control

The control problem consists of balancing two reasoning approaches to operating in an environment— deliberation and acting [Hanks & Firby 90]. Deliberation allows us to make many decisions ahead of time, thus making our choices more informed. However, information at execution time is typically more accurate than the projected future used during deliberation. Hence, by deferring the decision–making process to the last possible moment we may be able to make better decisions.

For the REA, this balance is achieved by splitting the deliberation and execution decision processes. Deliberation is performed by a planning agent external to the REA,

and execution decisions are made by the REA. This approach does not necessarily solve the "decision–theoretic control" problem [Boddy & Dean 89,Horvitz *et al.* 89], but does provide a framework such that the choice is not when to deliberate or when to act, but rather when does the deliberation system need to guide or assist the execution system.

For the research presented here, the role of the deliberation or planning system is to task the REA to achieve various activities in the environment which it is situated. In other words, it generates a plan that it has determined will achieve some desired effects. The REA's role is to decide how best to achieve the tasks given the execution environment and its knowledge and capabilities. The plan from the planning agent is used as a guide of how to select actions which together will lead to the desired effects.

In an effort to clarify all of this, consider how the REA is tasked by the planning agent. When the REA is first initialized it communicates with the planning agent and they exchange basic knowledge and capability information for use when deliberating. From that point the REA maintains a safe execution state (i.e., survives in the environment) until a task directive is received informing it which tasks it should carry out on behalf of the planning agent.

Once the planner has developed a plan it wishes the REA to carry out on its behalf, it packages that plan in the form of a IACL synthesize message (see Section 5.2.7). The message contains information such as ordering constraints on the actions, the teleology or causal structure related to the plan, temporal constraints, and commitment to its achievement. This information is then communicated to the REA.

The REA examines the type of communication event. The type informs the REA how it should process the information contained in the event—in this case it is of type *synthesize*. The REA posts the event to its agenda for processing by a capability for the particular type of message. The entry is selected from the agenda based upon its priority which is assigned according to the planning agent's commitment level and priority of the capability required to process it. When the "synthesize" agenda entry is selected it is processed and a task-directive is synthesized. During the synthesis process, the information contained in the synthesize is incorporated in the data structures of the REA. At this point the task-directive is placed back on the agenda to await execution.

The "execution" agenda entry is then selected from the agenda based upon its priority and commitment of the planning agent to its achievement. Once selected for execution, its monitors (which were established during the synthesis process from the causal structure information) become active and the actions are executed according to their ordering constraints. Execution continues until either all of the tasks of the task-directive have been achieved or a failure occurs. If execution was successful then the planning agent is so notified. Otherwise, the type of failure dictates how it is to be addressed. If it turns out that the REA is unable to address the failure with the tactics which are available to it, then it packages up information related to the failure and sends it to the planning agent to assist with recovery.

4.4.1 REA Control Mechanism

The control mechanism of the REA is not itself a single entity of the architecture. Instead, control is brought about by combining the functionality of several of the components of the architecture with the knowledge and capabilities available to the REA.

Control of task execution is primarily divided between the AM, TM, KP, and the capabilities (i.e., knowledge sources) Supervise, Execute, and Dispatch. The AM orders active intentions according to priority, the TM notifies the AM when additional intentions have become active, and the KP runs the knowledge sources that make sure that constraints and conditions are met for the execution of tasks in the environment. A task is placed on the agenda as an agenda entry (AE) for one of the capabilities Supervise, Execute, or Dispatch depending upon its present decomposition. High-level tasks contained in a task directive are processed by the Supervise capability. Primitive tasks are processed jointly by the Execute and Dispatch capabilities, and task networks are processed by the Execute capability.

In Section 4.4.2 we will consider an example to better understand how these components interact to control the REA's behavior.

4.4.2 Execution Example

As an example of how the TBL constructs, processing capabilities, and components of the architecture combine to control execution, consider the execution of the fly-transport task shown in Figure 4.4. We will forego the details of task directive synthesis for the moment[7], and begin our discussion as the Task Directive Object (which contains the fly-transport task) is received for processing by the Supervise capability. The fly-transport task is a high level description of activities that will move ground transports (i.e., gt1 and gt2) from City-K to the city of Delta on the island of Pacifica via the air cargo transport c5-1. Initially, the c5-1, gt1 and gt2 resources are located in City-K.

The Supervise capability is responsible for selecting tasks from the Task Directive Object (TDO) according to the algorithm given in Appendix B. Processing by the Supervise capability begins as the TDO enters with an execution status of "ready." The fly-transport task is specified in the TDO as:

(fly-transport c5-1 delta city-k)

which states that the c5-1 air cargo transport resource is to be flown to Delta from City-K. When cargo is not specified it defaults to ground transport resources. So, the task is to move ground transports gt1 and gt2 via the c5-1 resource from point A to point B.

During the processing of the TDO by the Supervise capability, it is determined that the fly-transport task is eligible to execute according to the ordering constrains of the TDO, and is cleared to execute since it has no conditions to be satisfied. At this point, a procedure must be selected which will perform the fly transport operation. This selection process is based upon the TBL *context* construct. For the fly-transport task, either the resource specified for the task is empty or loaded, is available for use and is at the location of origin. Assume for the sake of this example that the specified resource, c5-1, is empty and available; therefore, procedure fly-transport-1 is chosen. This procedure is then posted to the AM to be processed by the Execute capability. Since the FLY-TRANSPORT task is the only task cleared for execution in the TDO, execution of the remaining tasks of the TDO are suspended. At this point, the AM's

[7] The synthesis process will be described in detail in Chapter 5.

```
(make-instance 'Domain-Data
     :name 'fly-transport
     :expands '(fly-transport ?res ?to-loc ?from-loc)
     :procedures
      (list
       '(FLY-TRANSPORT-1 network
         (context (and (res-status '?res 'available)
                       (at '?res '?from-loc)
                       (load-status '?res 'empty)))
         (network ((tn1 (load ?res ?from-loc GTs-ONLY) ())
                   (tn2 (fly-plane-to-dest ?res ?from-loc ?to-loc) ())
                   (tn3 (act-sensor ?res) ())
                   (tn4 (unload ?res ?to-loc) ())
                   (tn5 (refuel ?res) ())))
         (ordering ((tn1 nil (tn2))
                    (tn2 (tn1) (tn3))
                    (tn3 (tn2) (tn4))
                    (tn4 (tn3) (tn5))
                    (tn5 (tn3) nil))))
       '(FLY-TRANSPORT-2 network
         (context (and (res-status '?res 'available)
                       (at '?res '?from-loc)
                       (load-status '?res 'loaded)))
         (network ((tn1 (fly-plane-to-dest ?res ?from-loc ?to-loc) ())
                   (tn2 (act-sensor ?res) ())
                   (tn3 (unload ?res ?to-loc) ())
                   (tn4 (refuel ?res) ())))
         (ordering ((tn1 nil (tn2))
                    (tn2 (tn1) (tn3))
                    (tn3 (tn2) (tn4))
                    (tn4 (tn2) nil)))))
     :effects '((at '?res '?to-loc)
                (load-status '?res 'empty))
     :end-cond '(and (>= (fuel-level '?res)
                         (* *air-threshold* (max-fuel '?res)))
                     (at '?res '?to-loc)
                     (load-status '?res 'empty)))
```

Figure 4.4: Fly Transport DDO

```
                    Agenda (cycle:    6)

<AE-5    :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>

------------------------ Untriggered Entries ------------------------
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
```

Figure 4.5: AM Agenda Snapshot in Cycle 6

agenda would appear as in Figure 4.5.

The AM, having been notified that the KP is idle, then selects an active intention to be processed. In this particular instance only one intention is active (i.e., AE-5) hence it is selected and sent to the KP. The KP checks if the required capability (i.e., Execute) is loaded, and if not, loads it.

The Execute capability is responsible for decomposing network procedures and posting primitive procedures for the Dispatch capability according to the algorithms given in Appendix C. The fly-transport-1 procedure is a network and has an execution status of "ready" since it has not been previously considered by the Execute capability.

We see, in Figure 4.4, that the fly-transport-1 network is composed of five subtasks — load, fly-plane-to-dest, act-sensor, unload, and refuel. The Execute capability locates Domain Data Objects (DDOs) in the REA's Domain Data Library corresponding to each subtask (according to the TBL *expands* construct), and synthesizes Procedure Objects for each of their procedures.

Once the Procedure Objects of each subtask in the network have been synthesized, Execute determines which subtasks are eligible to be executed. This is done according to the ordering and temporal constraints of the fly-transport-1 procedure. For each eligible subtask, Execute selects a contextually valid procedure and posts it to the AM for processing by Execute[8]. When all of the subtasks which are eligible to execute have

[8] At this particular point in the processing, Execute is not concerned whether the procedure it is posting is a primitive or a network. All it knows is that it has a procedure object that requires processing by the Execute capability. The issue here is whether to design knowledge sources so they execute in short periods allowing other capabilities the chance to run, or whether they have complex design and do not relinquish control until all possibilities have been considered. The former

```
                    Agenda (cycle:    7)

<AE-7   :  95 (EXECUTE #<PROCEDURE LOAD-2>)>

------------------------ Untriggered Entries-------------------------
<AE-8   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
```

Figure 4.6: AM Agenda Snapshot in Cycle 7

been posted, those which remain are suspended. They remain suspended until one of the posted subtasks executes successfully, or until a failure is detected. At this point in executing the fly-transport-1 procedure of the fly-transport task, the only subtask eligible to execute is load. For it, the load-2 procedure is chosen by Execute. With load-2 posted to the AM for Execute and the remaining subtasks of fly-transport-1 suspended, the agenda appears as that shown in Figure 4.6.

With the processing of the load-2 procedure by Execute we get our first look at the concurrent task execution capabilities of the REA. load-2 is a network task and has an execution status of "ready" as processing by the Execute capability begins. The load-2 network is:

```
'(LOAD-2 network
  (context (and (o-type '?res 'air-cargo-transport)
                (at '?res '?location)))
  (network ((tn1 (gt-sensor gt1) ())
            (tn2 (gt-sensor gt2) ())
            (tn3 (act-sensor ?res) ())
            (tn4 (load-cargo-plane ?res ?location ?cargo)
                 ())))
  (ordering ((tn1 nil (tn4))
             (tn2 nil (tn4))
             (tn3 nil (tn4))
             (tn4 (tn1 tn2 tn3) nil)))
```

Execute goes through the same process as before in locating DDOs and synthesizing procedure objects. However, when it comes to checking the ordering constraints of

approach was adopted, but whether this is the best approach remains an open research issue.

```
                    Agenda (cycle:    8)

<AE-9    :  95 (EXECUTE #<PROCEDURE GT-SENSOR-1>)>
<AE-10   :  95 (EXECUTE #<PROCEDURE GT-SENSOR-1>)>
<AE-11   :  95 (EXECUTE #<PROCEDURE ACT-SENSOR-1>)>

------------------------ Untriggered Entries------------------------
<AE-12   :  95 (EXECUTE #<PROCEDURE LOAD-2>)>
<AE-8    :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
```

Figure 4.7: AM Agenda Snapshot in Cycle 8

load-2 it finds that (gt-sensor gt1), (gt-sensor gt2), and (act-sensor c5-1) are all eligible to execute. It subsequently posts each of these subtasks to the AM for Execute and suspends the load-2 procedure until their completion. At this point, the agenda appears as that shown in Figure 4.7.

AE-9 is selected by the AM and the Execute capability begins processing the primitive procedure gt-sensor-1 which has an execution status of "ready." Execute determines if there is indeed a need for the task to be executed in the environment by querying the World Model to establish whether the effects of the task have already been realized in the environment. In this case they have not, so the gt-sensor-1 task is posted to the AM for the Dispatch capability. The static priority of the Dispatch capability is greater than that of the other two active intentions (i.e., AE-10 and AE-11), so the AM selects the intention that requires processing by Dispatch. The role of the Dispatch capability is to issue a task in the format appropriate for the type of hardware, simulator, or agent the REA is to be communicating. With the gt-sensor-1 task dispatched, the AM selects the next intention to be processed (i.e., AE-10).

AE-10 and AE-11 are processed in the same manner as AE-9 and their successful execution causes the triggering of AE-12. The load-2 network procedure is then ready to be processed by Execute again. This time, however, its execution status is that of "processing." The remaining subtask of the load-2 network is load-cargo-plane. For it, the load-cargo-plane-1 procedure is selected and posted to the AM for Execute. Once

```
                    Agenda (cycle:  152)

<AE-153 :  95 (EXECUTE #<PROCEDURE ACT-SENSOR-1>)>
<AE-154 :  95 (EXECUTE #<PROCEDURE UNLOAD-2>)>

------------------------ Untriggered Entries------------------------
<AE-155 :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
```

Figure 4.8: AM Agenda Snapshot in Cycle 152

the load-2 network has successfully executed, AE-8 becomes triggered so the remaining subtasks of the fly-transport-1 network procedure can be processed by Execute.

Of the four remaining subtasks of fly-transport-1, only the fly-plane-to-dest is eligible to be executed. Therefore, the fly-plane-to-dest-1 procedure is selected and posted to the AM for Execute. fly-transport-1 is again suspended and posted to the AM as the untriggered intention AE-83. fly-plane-to-dest-1 is a network procedure composed of the subtasks— taxi, get-clearance, liftoff-fly, and land. When each of these subtasks successfully completes the AM selects AE-83 for processing by the Execute capability. Execute determines that the subtasks act-sensor and unload are eligible to be executed concurrently and posts them to the AM. The remaining subtask, refuel-plane, is suspended as fly-transport-1 is again posted to the AM as AE-155[9] (Figure 4.8).

In cycle 202, fly-transport-1 is again processed by the Execute capability. The procedure refuel-plane-1 is chosen for the last remaining subtask, posted to the AM for Execute, and fly-transport-1 suspended. Refuel-plane-1 is a primitive with an execution status of "ready," and as such, Execute determines if it is necessary to post the task to Dispatch. In this instance, it determines that there is no need to execute the task and therefore, does not.

Now that each of the subtasks of fly-transport-1 has executed successfully, the AM selects AE-6 as the next intention to process in order to continue processing the re-

[9] Note that other tasks such as sensing by the ABM, reports of effects, etc., have been occurring, thus the cycle shown in the agenda has increased.

maining tasks of the TDO.

4.4.3 Selection Alternatives

There are three areas where the control mechanism must weigh alternatives. First, when the AM chooses a task to process next. Second, when a procedure of a task must be chosen. Third, when a Domain Data Object must be chosen from the REA's DDO Library.

Task Selection

The AM only manages active intentions and always selects the intention on the top of the agenda. When multiple intentions become active, the AM orders the intentions by the following criteria:

- Priority of the capability required to process the intention.
- Commitment level of the planning agent to the Task Directive to which the intention belongs.
- Temporal order of the intention on the agenda.

In executing a task, other factors determine which subtasks are posted to the AM so they too can become intentions of the REA. These are:

- Ordering constraints.
- Temporal constraints.
- Contextual constraints.

Ordering constraints dictate which tasks are eligible for execution by specifying those tasks that must have successfully executed previously. Temporal constraints force the execution of tasks to occur in temporal windows after the successful execution of other tasks. Contextual constraints specify the conditions which must be satisfied in the World Model before execution can take place.

Procedure Selection

Selection of a particular procedure of a task to bring about its desired effects is based upon two factors. These are:

- Contextual constraints.
- Order in the task.

Contextual constraints are used in the same manner as for task selection. That is, they specify the conditions under which a procedure is applicable. The second factor, order in the task, is applied when more than one procedure for a task is contextually

```
(make-instance 'Domain-Data
     :name 'task-xyz
     :expands '(task-xyz ?a ?b ?c)
     :procedures
      (list
       '(P1 network
         (context ...)
         (network ...)
         (ordering ...))
       '(P2 network
         (context ...)
         (network ...)
         (ordering ...))
       '(P3 network
         (context ...)
         (network ...)
         (ordering ...)))
...)
```

Figure 4.9: Task XYZ Procedure Orderings

valid. In this situation, the procedure that is specified higher in the task is chosen. For example, consider TASK-XYZ in Figure 4.9. Given procedures P2 and P3, both contextually valid, procedure P2 would be selected.

DDO Selection

Firby and others who have used the RAP representation have assumed that there is only one DDO for each task [Firby 89]. This does not have to be the case, and is not the case for the TBL representation used here.

The TBL allows a particular task to have multiple Domain Data Objects which describe various ways of achieving that task. This allows the REA to select the best possible method of achieving a task at runtime. The RAP approach allowed the user to specify various methods of achieving a particular task in the definition of a RAP too. However, the difference here is that by specifying multiple DDOs (or RAPs using Firby's nomenclature) for a task the REA can reason about things such as the cost difference between the methods, the number of preconditions which have to be satisfied, the number of effects or side-effects in using a particular approach, or the length of time one method will take versus another[10].

The REA uses a heuristic estimate of cost along with conditions, effects, and duration information in selecting between DDOs when more than one are appropriate to accomplish a task. The DDO with the lowest sum of the criteria ratings is selected first.

This approach provides the system with additional flexibility and more information to make better execution time decisions.

4.5 MAD - Design and Redesign

To achieve the integration of the capabilities required by the model, the underlying architecture must provide certain functionality. That is, it must be able to maintain multiple tasks with varying priorities. It must be able to easily change its processing behavior by adding or removing declarative knowledge of such behavior. It must possess a triggering mechanism to select appropriate behaviors based upon events and temporal deadlines. The architecture must also be able to accept asynchronous inputs, and

[10] This could be accomplished by modifying the TBL representation so that procedures within a DDO contained this information as opposed to having seperate DDOs, but the decision was made as a matter of personal preference.

be functionally independent of the underlying representation and functionality of the agent.

The O–Plan agenda–based reasoning architecture [Tate *et al.* 94] was selected as the base architecture for the REA since it appeared to provide all of the basic means to meet the requirements of this model. The research approach was to make use of this architecture as an initial basis, but to be prepared to adapt it or to use an alternative if unforeseen requirements were discovered.

4.6 Chapter Summary

In this chapter have discussed the motivation for the REA design, and the design itself. We examined in detail the five primary components which together make up the REA architecture: Communication Manager, Agenda Manager, Knowledge Platform, Trigger Manager, and Active Behavior Manager. Additionally, we discussed the REA's control mechanism and considered a detailed example to clarify that discussion.

In the next chapter, we will discuss the Inter–Agent Communication Language (IACL). This simple message protocol allows for the dynamic adaptation and modification or addition of knowledge and capabilities. It provides a flexible means to modify the execution time behavior of the REA thus, making it more robust in complex environments. This language allows a planning agent to task the REA to execute tasks on its behalf in the environment which the REA is situated.

Chapter 5

Inter-Agent Communication

As we saw in Chapter 4, the design of the REA is such that the planning agent and the REA are separate, concurrently operating asynchronous processes. Additionally, we saw that there was the requirement for the REA to acquire new knowledge from the planning agent in terms of the tasks it is able to carry out in the environment, and the requirement to acquire additional functionality to modify its execution behavior.

These requirements lead us to discuss the medium by which the REA is tasked to execute plans on the behalf of the planning agent and to modify its execution behavior. In this chapter we discuss the language by which information is transferred between the planning agent and the REA. This language not only allows the REA to be tasked, but allows adaptation of the REA's behavior so that novel situations in the environment can be addressed.

For complex and dynamic domains the set of possible interactions and situations the agent may find itself is practically infinite and cannot be specified beforehand. Therefore, we must make our agents adaptable when the specification of the environment and the agent's capabilities is incompatible, incorrect, or simply at an inappropriate level of abstraction [Kaelbling 91].

Section 5.1 discusses some of the issues related to communication. In Section 5.2 the Inter-Agent Communication Language (IACL) which allows the REA to communicate with the planning agent to receive new knowledge, capabilities and behavior modifications while situated in the environment is described. Section 5.3 describes the process of interpreting IACL messages, and Section 5.4 describes how new message types can

be dynamically created so that the REA can understand their content and adapt its behavior.

5.1 What is communication?

When humans communicate, we exchange messages. We send them, receive them, interpret them, and store or act upon them [Dimbleby & Burton 92]. Communication between computer–based agents is much the same. That is, as long as the agents involved in the communication understand the contents of the messages and share the same overall context.

But understanding involves far more issues related to communication. Issues such as the impact of communication on a recipient agent [Cohen & Levesque 90,Perrault 90], how much communication is necessary to get the desired beliefs and intentions ascribed [Pollack 86,Rosenschein 87,Gmytrasiewicz *et al.* 91], how to affect the behavior of another so that one's intentions are conveyed [Allen 79,Appelt 81,Konolige & Nilsson 80,Georgeff 83], and how to plan to communicate [Suthers 94,Turner 94,Lochbaum 94] are all important aspects of communication.

The motivation behind the design of a language for communication between the REA and the planning agent is to focus on how to use a language to task and modify the behavior of a reactive agent. However, in developing such a language and communication mechanisms, we do not want to limit others from addressing the more cognitive issues related to communication within a similar framework.

The primary design goals related to the communicative abilities of the REA are to:

- provide a language of typed messages that describes to the receiver how the contents are to be interpreted,
- provide a flexible mechanism for interpreting messages, and
- provide a flexible means to dynamically create new message types not already provided for in the language.

The last design goal was crucial to address the problems typically found in protocols. As [Durfee *et al.* 94] point out:

> When developing systems for accomplishing well–understood tasks in well–defined environments, it is reasonable and proper for a system designer to define and institute appropriate protocols. However, when the nature of agents' tasks might change, or their environment might undergo substantial changes, being locked into a particular interaction protocol might lead to ineffective action and interaction on the parts of the agents. Accomplishing tasks in such environments might require agents to invent new protocols based on experience and on expectations about how messages will affect each other.

The intent is that with IACL one would not have to invent a new protocol, but rather simply define the necessary message types to continue effective action. However, time will be the true test of this goal's achievement.

Each of these design goals are discussed in more detail in the following sections.

5.2 The IACL Protocol

The Inter Agent Communication Language is a common ontology that allows the REA and a planning agent to communicate about a domain of discourse without necessarily operating on a globally shared theory of plans. Though IACL is not as formally defined as is the best–known communication ontology, KQML[1] [Finin *et al.* 92], the design criteria for ontologies [Gruber 93] have been observed. These criteria are:

Clarity: An ontology should effectively communicate the intended meaning of defined terms.

Coherence: An ontology should sanction inferences that are consistent with the definitions.

Extendibility: An ontology should offer a conceptual foundation for a range of anticipated tasks, and the representation should be crafted so that one can extend and specialize the ontology.

[1] The primary function of the Knowledge Query and Manipulation Language (KQML) is for query processing in databases and knowledge based systems, but could possibly subsume IACL using the KQML construct PACKAGE where the IACL message types were specified as content messages. However, it is believed that incurring that overhead would not yield significant gains in the research being studied here. The interested reader is directed to neches@isi.edu for a current copy of the ever changing KQML Specification documentation.

Minimal encoding bias: The ontology should be specified at the knowledge level without depending on a particular symbol-level encoding.

Minimal ontological commitment: An ontology should make as few claims as possible about the world being modeled, allowing the parties committed to the ontology freedom to specialize and instantiate the ontology as needed.

IACL is presently defined by nine message types. Each message type will be discussed in the following sections. We use the theme of the REA being a reconnaissance satellite to motivate the examples.

5.2.1 Add Active Behavior

The *add-active-behavior* type message is used to install new active behaviors to those already known to the REA. As will be seen in Chapter 6, the REA has two basic types of behavior—active and passive. Active behaviors are those which become active on a specific frequency. The add-active-behavior message, Figure 5.1, contains:

```
(:add-active-behavior
   (name          )
   (property      )
   (frequency     )
   (behavior      )
   (event         )
   (capability    ))
```

Figure 5.1: Empty IACL Add-Active-Behavior Message

- the name of the active behavior to add,
- the property that is to be affected by the activation,
- the period (in seconds) between activations,

- the type of behavior that is to be activated if the behavior is a sensory one,
- the event handler (i.e., REA capability) that is to be notified upon the behavior's activation, and
- the capability to install if the capability specified as the event handler is not already known by the REA.

When the *add-active-behavior* message is received by the REA, the leftin Guard checks to see if the event handler is currently known to the REA. If it is, then the message gets processed. Otherwise, the Guard makes sure that a handling capability is specified in the capability field of the message. If the event handler is unknown to the REA and no capability is specified then the Guard rejects the message by sending an *unknown-capability* message to the planning agent.

If the message is processed then the active behavior specified in the message is installed and becomes activated at the time of installation plus the behavior's frequency.

5.2.2 Add PS Behavior

The *add-ps-behavior* type message is used for one of two purposes; either to install a new passive behavior to those already known to the REA, or to install a new specialist behavior. Passive behaviors are opportunistic in nature and only become "active" when the conditions exist to trigger the behavior. A specialist behavior is not a particular type of behavior, but by being specified in an *add-ps-behavior* message it informs the REA that it is to install a new passive behavior, as well as a new capability which is also specified in the message.

The *add-ps-behavior* message, Figure 5.2, contains:

- the type of behavior (either passive or specialist) to install,
- the name of the passive behavior to add,
- the type of situations the behavior opportunistically observes,
- the trigger that determines the conditions under which the behavior becomes activated, and

```
(:add-ps-behavior
   (type          )
   (name          )
   (handles       )
   (trigger       )
   (action        ))
```

Figure 5.2: Empty IACL Add-PS-Behavior Message

- the action that is to be taken upon activation of the behavior. This is a specification of the REA capability that is to handle the behavior's activation.

A passive behavior is added to the REA's set of behaviors to monitor specific changes to the REA's World Model specified by the behavior's trigger. For example, we may want the REA to inform the planner when all of the pictures of a planet have been taken so the planner could task the REA to transmit them back to Mission Control.

The specialist behavior gives the REA the ability to adapt to its environment. For example, if while executing a plan the REA sent a failure message back to the planning agent because it had no knowledge of what to do when film in the camera controlled by the REA would not advance. We could send the REA an *add-ps-behavior* message with a passive behavior and a capability (Figure 5.3). The behavior would opportunistically monitor for the "no advance" problem and the capability could provide a means, say rewind and load new film, so that in the future the REA could address the "no advance" problem without failing to the planning agent thus, adapting the execution behavior of the agent. This example assumes that the REA possesses the knowledge to rewind and load the camera. If this was not the case, then *add-capability* messages (Section 5.2.3) giving the REA this knowledge would have to be sent prior to this *add-ps-behavior* message. Otherwise, the REA would fail due to lack of knowledge.

5.2.3 Add Capability

The *add-capability* type message specifies a capability that is to be installed for use by the REA. If the capability specified in the message is already known to the REA then the capability in the message subsumes the old definition. The *add-capability* message,

```
(:add-ps-behavior
 (TYPE :specialist)
 (BEHAVIOR
  (NAME camera-film-no-advance)
  (HANDLES (no-advance))
  (TRIGGER
     (:WAIT-ON-EFFECT
        (and (status-camera '?res '?cstatus)
             (eq '?cstatus 'no-advance))))
  (ACTION (:camera-failure (effects nil film-no-advance))))
  (CAPABILITY
    (NAME :KS-CAMERA-FAILURE)
    (PRIORITY 20)
    (KS (defun KS-CAMERA-FAILURE (event)
          (let* ((*package* (find-package :rea))
                 (world-msgs (second (ag-body event)))
                 (data-type (first world-msgs))
                 (reason (third world-msgs)))
           (dbg :ks-camera-failure "**************************~%")
           (dbg :ks-camera-failure "Receiving failure event...~%")
           (case reason
             (FILM-NO-ADVANCE
               (let ((problem-camera nil)
                     (problem nil))
                (mapcar #'(lambda (camera)
                            (if (eq 'no-advance
                                    (status-camera camera))
                              (progn
                               (setq problem-camera camera)
                               (setq problem 'film-no-advance))))
                  *cameras*)
                (let
                  (owner (make-t-directive-obj
                          (gentemp "T-DIRECTIVE-")
                           'u nil 0 nil nil))
                  (parent nil)
                  (eligible-DDOs
                        (select-data-object
                          (find-DDO-with-pattern
                             '(rewind-and-reload ?res))))))
                 .
                 .
                 .))
        (REQD-FNS))))))
```

Figure 5.3: IACL Add-PS-Behavior Message

```
(:add-capability
   (name          )
   (priority      )
   (KS            )
   (reqd-funs     ))
```

Figure 5.4: Empty IACL Add-Capability Message

Figure 5.4, contains:

- the name of the capability to install,
- the priority level of the capability in relation to others,
- the code defining the capability, and
- any code necessary to support the capability's functionality.

This message type is used to install additional functionality or to modify the functionality of the REA dynamically. For example, if we wanted the REA to change course to take pictures of a comet that was going to pass nearby then we could dynamically add the capability to achieve this in the event that such a capability was not already available to the REA.

5.2.4 Add Knowledge

The *add-knowledge* type message allows us to add or modify the REA's knowledge of the tasks it can execute in the domain. By sending an *add-knowledge* message to the REA we specify a new domain data object that it can use for specifying the execution time behavior of a particular future task. The *add-knowledge* message, Figure 5.5, contains:

- the name of the knowledge to add or modify,
- the "expands" pattern that will trigger the use of this knowledge,
- the conditions under which the use of this knowledge is allowed,

```
(:add-knowledge
   (name            )
   (expands         )
   (conditions      )
   (effects         )
   (uses-resources  )
   (exec-cost       )
   (end-cond        )
   (duration        ))
```

Figure 5.5: Empty IACL Add-Knowledge Message

- the primary effects that can be expected to be satisfied in the environment upon successful completion of applying this knowledge,
- the resources that are used during the application of this knowledge,
- a heuristic estimate of the cost of applying this knowledge,
- the conditions under which application of this knowledge may be deemed successful, and
- an estimate of the earliest and latest finish time required to apply this knowledge.

If we determined a new way for the REA to change the position of its large radar array such that it required less battery power, we would want the REA to use that new technique every time we tasked it to change the position of the array. The *add-knowledge* message would allow us to do just that.

5.2.5 Execution Failure

The *execution-failure* type message is used by the REA to inform the planning agent that one of seven types of failure has occurred. These are to notify the planning agent that a particular plan that the REA was attempting to execute has failed in some way and thus, the REA is providing information necessary to the planning agent so it can initiate some solution or simply be aware of the problem. When a failure occurs that is beyond the failure–handling knowledge and capabilities of the REA, an *execution-*

failure message is sent to the planning agent and all information related to the failed plan is relinquished by the REA.

```
(:execution-failure
   (failure-type       )
   (affected-nodes     )
   (affected-cstrs     )
   (affected-resources )
   (reason             ))
```

Figure 5.6: Empty IACL Execution-Failure Message

The *execution-failure* message, Figure 5.6, contains:

- the type of failure that occurred,
- the plan nodes that were involved in the failure (i.e., the nodes which were executing at the time of the failure),
- the causal structure record tags that were active active at the time of the failure,
- the resource(s) that were affected by the failure (i.e., those resources being used by the task that failed), and
- the reason for the failure (if one could be determined).

Chapter 6 will discuss how failures are detected and how the *execution-failure* message is used in such situations.

5.2.6 Remove Capability

The *remove-capability* type message is the converse of the *add-capability* message (Section 5.2.3). It informs the REA that the capability specified in the message in no longer required and that it is to remove all internal knowledge of that capability.

The *remove-capability* message, Figure 5.7, is a simple message that contains the name of the capability to be removed. When the *remove-capability* message is received by the REA, the leftin Guard checks to see if the capability is currently known to the REA.

```
(:remove-capability
   (capability          ))
```

Figure 5.7: Empty IACL Remove Capability Message

If it is, then the message gets processed. Otherwise, the Guard rejects the message by sending an *unknown-capability* message to the planning agent.

5.2.7 Synthesize

The *synthesize* type message is the means by which the planning agent communicates a plan to the REA which is to be carried out on its behalf, thus tasking the REA to execute the plan. It contains the information necessary for the REA to be able to execute actions, possibly in parallel, and to be able to monitor important aspects during the execution of the plan.

```
(:synthesize
   (node-network )
   (priority     )
   (orderings    )
   (cstr         ))
```

Figure 5.8: Empty IACL Synthesize Message

The *synthesize* message, Figure 5.8, contains:

- the actions of the plan (i.e., tasks to be carried out),
- the commitment level of the planning agent to the achievement of the plan,
- the constrains on the orderings of the actions, and
- the causal structure generated during the planning process specifically related to the actions of the plan.

The final result of receiving and processing a *synthesize* message is a Task Directive

Object (Section 3.2.1) which represents the information of the *synthesize* message in a form that allows the REA to execute tasks.

When a *synthesize* message is received by the REA, the leftin Guard determines whether the REA has the knowledge to carry out each of the tasks of the plan. If not, an *unknown-knowledge* message (Section 5.2.9) is sent to the planning agent to inform it that it has requested the REA to execute a task for which it has no knowledge. If the REA does have the necessary knowledge then the message is sent to the Agenda Manager (Section 4.3.3) for further processing.

In Section 5.3 we will examine how the IACL message types are interpreted and specifically discuss how the information contained in the *synthesize* message is used to synthesize Task Directive Objects.

5.2.8 Unknown Capability

The *unknown-capability* type message is used by the REA to inform the planning agent that it does not possess a particular capability that was referred to in an IACL message. The sending of this message type is initiated by the leftin Guard before the REA has a chance to process the contents of the message.

```
(:unknown-capability
   (capability          ))
```

Figure 5.9: Empty IACL Unknown Capability Message

The unknown-capability message, Figure 5.9, only contains the name of the capability which is not known by the REA.

5.2.9 Unknown Knowledge

The final message type in the current IACL Specification is that of *unknown-knowledge*. The *unknown-knowledge* message is used for a similar purpose as that of the *unknown-capability* message, but it refers to knowledge about a task which the REA does not possess.

```
(:unknown-knowledge
   (knowledge          ))
```

Figure 5.10: Empty IACL Unknown Knowledge Message

The *unknown-knowledge* message, Figure 5.10, is again a simple message that only contains the name of the task which is not known to the REA.

5.3 IACL Message Interpretation

The type of an IACL message specifies how a particular message is to be processed and thus, interpreted. The REA must possess a capability for each IACL message it is to receive. When the Communication Manager (Section 4.3.2) receives a communication event on the leftin channel, the Guard examines the type to determine if it should reject or accept the message. Rejection occurs in three instances. First, if the type of message is not known by the REA (i.e., the REA does not possess a capability to process the message contents) then an *unknown-capability* message is sent and no further processing takes place for that message. Second, if the message is one of *add-active-behavior*, *add-ps-behavior*, or *remove-capability* and the specified capability is not known by the REA then an *unknown-capability* message is sent and no further processing takes place for that message. Finally, if the message type is synthesize and the REA does not possess the knowledge to execute a particular task in the plan it is to execute, an *unknown-knowledge* message is sent and no further processing takes place for the message. If the message is not rejected in one of these three cases then it is accepted and passed to the Agenda Manager (Section 4.3.3) for further processing.

When the message is converted to an agenda entry the type of message specifies the capability required to interpret the contents of the message and perform the appropriate action(s). Once the Agenda Manager selects the agenda entry that contains the message it is passed to the Knowledge Platform (Section 4.3.4), the appropriate capability is loaded (if not already loaded), and the message is interpreted and processed by the capability.

```
(:synthesize
  (node-network
    ((NODE-3 (FLY-TRANSPORT C5-1 Delta City-K) ())
     (NODE-4-1 (DRIVE GT2 Abyss Delta) ())
     (NODE-4-2 (LOAD GT2 Abyss PASSENGERS) ())
     (NODE-4-3 (DRIVE GT2 Delta Abyss) (DRIVE-1))
     (NODE-4-4 (UNLOAD GT2 Delta) ())
          .
          .
          .

     (NODE-8 (FLY-TRANSPORT C5-1 City-K Delta) ())))
  (priority 1)
  (orderings
    ((NODE-3 nil (NODE-5-1 NODE-6-1))
     (NODE-4-1 (NODE-5-4) (NODE-4-2))
     (NODE-4-2 (NODE-4-1) (NODE-4-3))
     (NODE-4-3 (NODE-4-2) (NODE-4-4))
     (NODE-4-4 (NODE-4-3) (NODE-7))
          .
          .
          .

     (NODE-8 (NODE-4-4 NODE-6-4) nil)))
   (cstr
     ((CSTR-1  (AT GT1) DELTA (NODE-6-1) (NODE-3))
      (CSTR-6  (RES-STATUS GT2) DRIVING (NODE-4-2) (NODE-4-1))
      (CSTR-7  (AT GT2) ABYSS (NODE-4-2) (NODE-4-1))
      (CSTR-9  (AT GT2) DELTA (NODE-4-4) (NODE-4-3))
      (CSTR-10 (AT GT2) DELTA (NODE-8) (NODE-4-3))
      (CSTR-11 (AT GT2) DELTA (NODE-8) (NODE-4-4))
      (CSTR-15 (AT GT2) DELTA (NODE-4-1) (NODE-5-3))
      (CSTR-17 (RES-STATUS GT2) AVAILABLE (NODE-4-1) (NODE-5-4))
      (CSTR-23 (AT GT1) DELTA (NODE-8) (NODE-6-4)))))
```

Figure 5.11: Partial IACL Synthesize Message

To better understand this process, let's look at an annotated execution trace to examine the synthesis process in more detail and see what exactly happens as the contents of a *synthesize* message are interpreted. Take for example, the *synthesize* message shown in Figure 5.11. The first thing that occurs is that for each task specified in the node-network, a schema object (Section 3.2.3) is synthesized from that information and a Domain Data Object (DDO) from the REA's library. The node-network information has the form:

(node-reference (task-name args) (preference-constraints))

The *node-reference* allows the planner and the REA to communicate about a particular task in a plan even thought their representations of that task are different[2]. The *task-name* and *args* specify the knowledge (i.e., DDO) that is required to define the execution time behavior of the task. Lastly, the *preference-constraints* allow the planner to limit the knowledge that may be used to achieve a particular task. For example, if the REA had three DDO definitions that would allow it to achieve a task of moving something from one place to another, say by pack-mule (*move*), truck (*move-1*), or helicopter (*move-2*), the planner could constrain that selection. The planner might wish to limit the moving by land thus preferring *move* or *move-1*, but not *move-2*. Such a constraint could be specified as:

(node-xyz (move apples a b) (move move-1)

The task-name and args (together called the pattern) are used to to find all appropriate DDOs whose variables unify with the pattern. Once the DDO from the library is located, the schema object is synthesized.

```
:KS-SYNTHESIZE Receiving Synthesize Message...
:BUILD In find-DDO-with-pattern: (FLY-TRANSPORT C5-1 DELTA CITY-K)
:BUILD DDO-match : ((?FROM-LOC . CITY-K)
                    (?TO-LOC . DELTA) (?RES . C5-1))
:BUILD Exiting Find-DDO-with-pattern having found:
       (#<DOMAIN-DATA FLY-TRANSPORT>)
:BUILD SYNTHESIZE-OBJECT: matching-objects is
       (#<DOMAIN-DATA FLY-TRANSPORT>)
:BUILD      (NODE-3 (FLY-TRANSPORT C5-1 DELTA CITY-K) NIL)
:BUILD      Selected objects are (#<DOMAIN-DATA FLY-TRANSPORT>)
:BUILD      Synthesized #<SCHEMA NODE-1.0 FLY-TRANSPORT>
:BUILD In find-DDO-with-pattern: (DRIVE GT2 ABYSS DELTA)
```

[2] This reference information is stored in a cross-reference table internal to the REA and referenced whenever the REA must communicate information which concerns a particular task.

```
:BUILD DDO-match : ((?FROM-LOC . DELTA)
                    (?TO-LOC . ABYSS) (?RES . GT2))
:BUILD Exiting Find-DDO-with-pattern having found:
       (#<DOMAIN-DATA DRIVE>)
:BUILD SYNTHESIZE-OBJECT: matching-objects is (#<DOMAIN-DATA DRIVE>)
:BUILD      (NODE-4-1 (DRIVE GT2 ABYSS DELTA) NIL)
:BUILD      Selected objects are (#<DOMAIN-DATA DRIVE>)
:BUILD      Synthesized #<SCHEMA NODE-2.0 DRIVE>
```

After all of the schema objects have been synthesized from the node-network information, the next thing that occurs during the synthesis process is the use of the causal structure information to synthesize protection monitors. How these monitors are used is described in detail in Chapter 6, but for now we will focus on the process of establishing the monitors.

```
.
.
.
:MONITORS Trying to add monitoring info for NODE-4-1
:MONITORS Setting monitor on NODE-4-1 for
          (RES-STATUS GT2) = AVAILABLE from (NODE-5-4)
:MONITORS Setting monitor on NODE-4-1 for
          (AT GT2) = DELTA from (NODE-5-3)
:MONITORS Trying to add monitoring info for NODE-4-2
:MONITORS Setting monitor on NODE-4-2 for
          (AT GT2) = ABYSS from (NODE-4-1)
:MONITORS Establishing behavior for updates to (AT GT2).
:MONITORS Setting monitor on NODE-4-2 for
          (RES-STATUS GT2) = DRIVING from (NODE-4-1)
:MONITORS Establishing behavior for updates to (RES-STATUS GT2).
:MONITORS Trying to add monitoring info for NODE-4-4
:MONITORS Setting monitor on NODE-4-4 for
          (AT GT2) = DELTA from (NODE-4-3)
.
.
.
:MONITORS Trying to add monitoring info for NODE-8
:MONITORS Setting monitor on NODE-8 for
          (AT GT2) = DELTA from (NODE-4-4)
:MONITORS Setting monitor on NODE-8 for
          (AT C5-1) = DELTA from (NODE-3)
:MONITORS Establishing behavior for updates to (AT C5-1).
```

Each causal structure record from the IACL synthesize message has the form:

(tag (attrib obj) value (r-nodes) (c-nodes)

where *tag* allows the REA and the planning agent to reference specific protection intervals, *attrib, obj, value* specifies the entity in the environment whose attribute should have value from the last c-node to the last r-node. That is, c-nodes are those tasks in the plan whose effects yield the expected value for obj, and r-nodes are those tasks that require the value. A monitor is an object used to monitor changes to the REA's World Model and detect deviations of values from expected values at specified points during the execution of a plan. Monitors are only established if there is causal structure related to a particular task.

Once all of the monitors have been established, the remaining information of the *synthesize* message along with the newly created schema objects, are placed into a Task Directive Object.

```
 0~00:15:18...T-DIRECTIVE-1 has been synthesized.  It looks like:
Name:     T-DIRECTIVE-1
Status:   U
Priority: 15
Processing: NIL
Processed: NIL
Task:
 Name: NODE-3(1.0)          I-type: FLY-TRANSPORT      E-Status: U
 Name: NODE-4-1(2.0)        I-type: DRIVE              E-Status: U
 Name: NODE-4-2(3.0)        I-type: LOAD               E-Status: U
 Name: NODE-4-3(4.0)        I-type: DRIVE              E-Status: U
 Name: NODE-4-4(5.0)        I-type: UNLOAD             E-Status: U
 .
 .
 .
 Name: NODE-8(15.0)         I-type: FLY-TRANSPORT      E-Status: U
:KS-SYNTHESIZE ****** END OF KS-SYNTHESIZE ******
```

With the Task Directive synthesized, it is sent to the Agenda Manager to await further processing by the Supervise capability, and processing of the IACL *synthesize message* is complete.

5.4 Dynamic Message Creation

Dynamic message creation is a similar process to that of modifying an existing capability or adding a new capability. If one wanted to dynamically modify the way a particular type of IACL message was interpreted and/or processed then one would have to send an *add-capability* message that contained the modified or new capability. That way, all subsequent messages of that type addressed by the capability would be handled in the new or modified fashion.

Let's consider the REA as a satellite one last time to see how this would work. For the sake of argument, say that we wanted to determine the status of a particular plan's execution that we had previously tasked the REA to execute, and for some reason we did not think beforehand to include such a capability. First, we would need to provide the REA with the capability to process our "status" message. We would need to implement a capability that would be able to collect the necessary information and then return that information. Then we would need to send an IACL *add-capability* message that contained our newly created capability to the REA. Second, we would need to send our new IACL status message to the REA. It would then be a matter of waiting for the REA to process the message and return the results.

A more complex example would involve the use of new knowledge which would also have to be sent to the REA. The complexity would be a result of more communication steps necessary to achieve our goal and possibly a more complex capability to process the message. However, the process would be basically the same.

5.5 Chapter Summary

We have seen how a relatively simple inter–agent communication language provides powerful mechanisms for adaptability, flexibility, and robustness. The Inter–Agent Communication Language provides nine primary message types that allow information to be communicated between the REA and the planning agent in a common ontology.

The message types of IACL allow for the modification and/or creation of both knowledge and capabilities dynamically in the REA at runtime.

In the next chapter, we will see the failure management capabilities of the REA in detail. These capabilities include how monitors and causal structure are used to detect potential failures, and how active and passive behaviors are used to update the World Model and detect failures.

Chapter 6

Experimentation Testbed and Evaluation Criteria

In the previous five chapters we have discussed (1) a set of characteristics necessary for rational behavior in a complex and dynamic environment, (2) the architecture of an agent that uses this set as the fundamental basis of its design, and (3) a communication language that provides a means to dynamically adapt the behavior of that agent. Now we are ready to discuss how an agent based on this design can indeed behave rationally in a complex and dynamic environment.

In order to get to the point where we can compare implementations of the characteristics of Chapter 2, we need to apply them to the same problems under the same conditions. Though simulation does not provide the ultimate convincing results as say, the control of an autonomous robot (since success in simulation does not guarantee success in the real world or vice versa), it does provide a good place to begin. Hopefully, the simulator described in this chapter will provide a common ground for such comparisons.

In this chapter we describe the Pacifica Simulator that allows us to test a series of expectations about the REA's design to determine whether it actually performs as claimed, and demonstrate the various features and capabilities provided for in the design. This testbed allows us to observe how the design, which incorporates the characteristics from Chapter 2, will perform in a complex and dynamic environment. The Pacifica Simulator serves as a simplified, simulation version of just such an environment.

We begin with an overview of the testbed itself, discussing its characteristics, the underlying reasoning behind its development, and the environment it models. In Section 6.2 we discuss the interface of the simulator, the types of exogenous events that can occur in the environment, the objects modeled in the environment, and how the REA and the simulator interact. In Section 6.3 we discuss the specific scenarios that will be used to exercise the design, and finally we again relate the work to the MAD methodology to consider the expectations of the REA's behavior and the criteria for evaluation of the design.

6.1 Testbed Overview

The REA has been designed to be responsible for coordinating the activities of multiple resources under its control to perform various tasks in a command and control fashion. Hence, the testbed must allow for features (and the testing of capabilities) beyond those which have been used previously to test similar designs [Firby & Hanks 87,Pollack & Ringuette 90,Vere & Bickmore 90,Engelson & Bertani 92,Nguyen *et al.* 93]. These features include:

- remote sensing,
- complex object interactions,
- continuous time,
- dynamic reporting of task completion or failure,
- asynchronous task execution, and
- extended, probabilistic task durations.

To meet these requirements, I have designed and developed the Pacifica Simulator. The Pacifica Simulator is a testbed for studying transportation planning/scheduling, logistics, and Non-combatant Evacuation Operation (NEO) scenarios. It draws upon data from the PRECiS Environment [Reece *et al.* 93] to define a fictional domain that is a suitably realistic for studying such scenarios. Scenarios in this domain are based on a hypothetical theater of operations called the island of Pacifica.

The PRECiS (Planning, Reactive Execution, and Constraint-Satisfaction)[1] Environment defines the data and hypothetical background for studying logistics and transportation planning/scheduling problems and Non-combatant Evacuation Operations (NEO) scenarios. The definition of the PRECiS environment has drawn on work by: Brown at Mitre Corporation to describe a realistic NEO scenario for the Planning Initiative IFD2[2]; Reece and Tate at the University of Edinburgh to define a fictional environment suitable for demonstration and evaluation for planning and reactive execution of plans based on the island of Pacifica ([Reece & Tate 93]); and work by Hoffman and Burnard at ISX Corporation to produce a cut-down demonstration scenario suitable for transportation scheduling research experiments as part of the Advanced Research Project Agency/Rome Laboratory Planning Initiative.

The Pacifica simulator has been designed to show how an execution agent is able to handle simultaneous tasks which take place over extended time intervals and which may possibly interact. Pacifica models a command and control environment in which the execution agent takes the role of a field commander receiving mission directives from a superior agent and issuing orders to entities under its direct control to carry out those missions. These missions are in the context of NEOs. Such operations are undertaken to provide rapid response to a variety of circumstances such as storms and other natural disasters, evacuation of civilians from trouble zones, policing and medical missions, humanitarian relief, etc. They are often characterized by the need for rapid deployment of equipment and personnel to ensure effective aid is offered and to seek to minimize the escalation of the problems through delays. The transportation logistics associated with NEOs present many interesting problems for reactive execution.

The primary reason for the use of the Pacifica Simulator is to demonstrate the early failure detection, resource reasoning, reflexive knowledge, communication, change-of-focus, asynchronous input, and failure management capabilities of the REA design.

---

[1] A précis is a short piece of writing which contains the main points of a book or report, but not the details.

[2] Integrated Feasibility Demonstration (IFD) Number 2 is one in a series of demonstrations using data only available to Advanced Research Project Agency/Rome Laboratory Planning Initiative participants.

6.1.1 Theater Geography

The simulation environment relates to events which are to take place in a hypothetical theater of operation. This theater is located in the Pacific Ocean and consists of the island of Pacifica and a friendly Country-X which has an airport and seaport in City-K.

Pacifica (see Figure 6.1) is an island state located in the Pacific Ocean. It has a very interesting coastline, but remains shrouded in mystery due to its inaccessibility over the centuries with some areas of the island largely unexplored and unmapped. The island was formed by volcanic activity and still has one active volcano. There are active geothermal areas on the Western part of the island with volcanic mud occasionally closing the coastal island road for days at a time. A large fresh water lake has formed in a dormant volcano in the North, and prevailing winds come over the cliffs from the Northeast. The Southern portion of the island consists of the lush, tropical, Abysian Forest, and cotton is grown in the South-Central region. The small fishing village of Exodus is located on the Southeastern tip of the island, and its access is by what can only be described as a trail which limits the types of vehicles that can enter the village. The remainder of the island terrain consists mainly of a mixture of low growing shrub and vegetation. Typically, monsoons occur during the periods of January–February and July–August.

Pacifica has two seaports and airports. A seaport and airport are located in both the capital Delta and the city of Calypso.

6.2 The Pacifica Simulator

The Pacifica Simulator is a message–based discrete event simulator which features exogenous change, complex interaction among objects in the environment, uncertainty, sensing, and continuous time.

The simulator consists of:

- User interface with three primary windows (i.e., Pacifica Simulation History, Pacifica Events, and Simulator Interaction) and one window for displaying the current simulated time (i.e., Pacifica Clock).

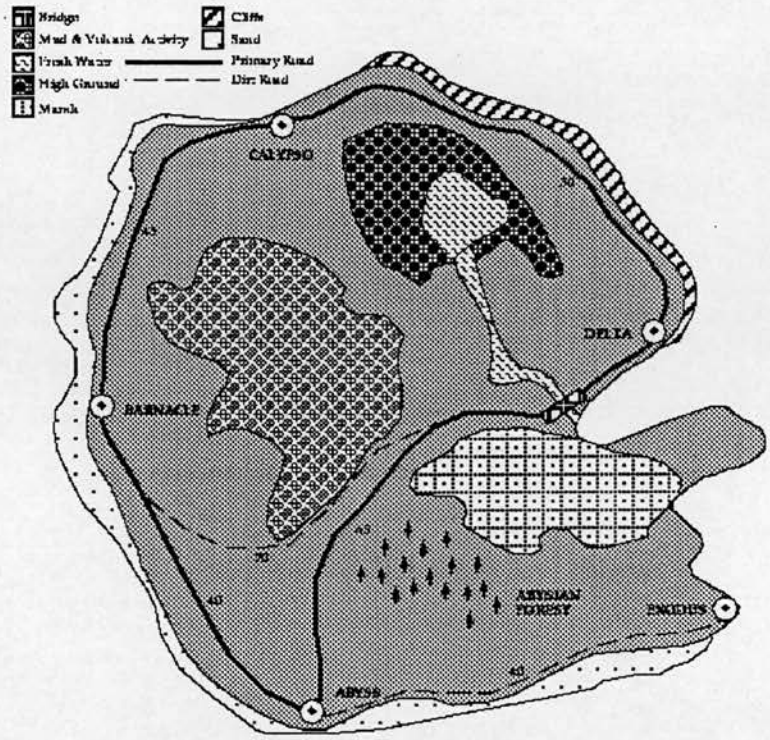


Figure 6.1: Island State of Pacifica

- Common Lisp Object System (CLOS) object definitions which define the entities of the world and their characteristics.
- Four exogenous event types (i.e., Meteorological, Resource, Terrorist-activity, and Natural-disaster) that occur probabilistically.

The Pacifica Simulation History window displays a summary of what has happened in Pacifica since the simulation began. The Pacifica Events window shows the events which are scheduled to happen and have yet to occur, and the Simulator Interaction window allows the simulation user to intervene to cause events to fail, be delayed, or occur sooner than scheduled (see Figure 6.2 for a sample run of Pacifica). The simulator is also capable of producing a snapshot file on command which details modeled characteristics for display with a graphical viewer (e.g., O-Plan's World Viewer).

Events occur either as a result of receiving a message to initiate a particular action in the environment or by the Exogenous Event Manager (EEM). The EEM generates random events based upon user–defined probability distributions. The simulator pro-

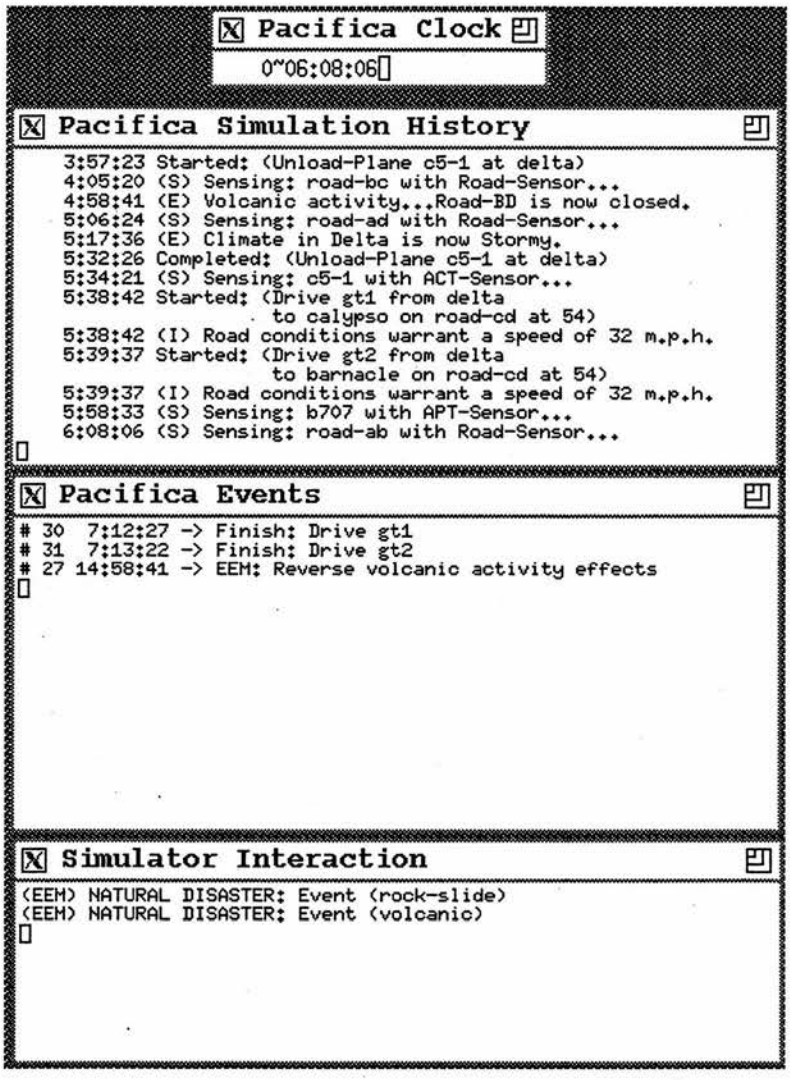


Figure 6.2: Pacifica Simulator

vides an open ended framework for easily adding such events. However, at present, four types of exogenous events can occur during the simulation. These are meteorological events, resource events, terrorist activity, and natural disasters. The following shows the events built into the simulator for the purposes of the evaluation of this research.

Meteorological events change the weather at randomly selected cities in Pacifica and subsequently effect the road conditions between cities. Road conditions along with road types determine how fast a ground-transport may travel on a particular road. The effects of meteorological events are to change the climate at a particular location to one of "sunny," "rainy," or "stormy."

Resource events effect characteristics of transportation resources such as fuel levels, mechanical status, and condition of tires. If the fuel level of a particular resource is below a user–defined threshold when the resource is to be used then this results in a failure. The mechanical status of a resource must be "good" and the condition of the tires must be "good" before a resource can be used. Otherwise, an appropriate failure is signaled to the REA. The effects of resource events can be no change to the resource, reduction of the resource's fuel level by 10%, reduction of fuel to zero, a change in the resource's mechanical status to one of "good," "poor," or "bad," or a change in the resource's tire status to "okay" or "blown."

Terrorist activity events effect ground transport resources and bridges. At present, terrorist events only occur on ground transports if the ground transport is found to be traveling down a road where a terrorist group is currently located. If attacked by terrorists, ground transport resources can have their mechanical status reduced to "poor" or "bad", tires shot out, or even be captured. However, they may also be lucky enough to escape unharmed.

The fourth exogenous event type is Natural disaster. These events take the form of rock slides, volcanic activity, and floods. Rock slides slow down ground transport resources by 10% on the affected road. Volcanic activity closes roads and can potentially destroy ground transports if the ground transport is in a specific five mile stretch of road. Floods simply close roads. The effects of rock slides only affect a ground transport if on the road at the time and does not affect subsequent travel along the road. Volcanic and flood events can affect roads for long periods of time. The effects of volcanic activity lasts for 10 hours unless another volcanic event occurs during that 10 hour period. If another volcanic event occurs before the 10 hour period has expired then an additional 3 hours is added to the time before the road re–opens. Floods last for 5 hours unless another occurs in which case 2 more hours are added before the road is open for travel again.

6.2.1 Modeled Characteristics in Pacifica

The Pacifica Simulator currently models entities related to the NEOs defined in the PRECiS Environment [Reece *et al.* 93]. That is, it models resources, geographical

objects, and the entities related to the exogenous event types. The individual characteristics modeled depend on the type of entity. For instance, ground-transport resource objects have thirteen dynamic and ten static characteristics, while bridge objects only have two (one dynamic and one static) characteristics which are modeled in the Pacifica Simulator.

Tasks

Twenty-three task types are presently accepted by the Pacifica Simulator. These range from simple sensor requests to complex drive requests which can have any one of seven preconditions not satisfied which result in failures.

Resources

Five types of resources are modeled in the Pacifica Simulator. These are:

- Ground Transports,
- Air Cargo Transports,
- Air Passenger Transports,
- Medical Helicopters, and
- Fast Sealift Ships.

Geographical Objects

Modeling geographical entities on the island of Pacifica include:

- Cities,
- Bridges,
- Roads,
- Airports, and
- Seaports.

Exogenous Event Entities

The Exogenous Event Manager determines if, when, and what type of an exogenous event should occur during the simulation. In addition to manipulating the values of other modeled objects in Pacifica, it controls the presence of:

- Terrorist Groups,
- Weather, and
- Natural Disasters.

6.2.2 Simulator Interaction

The interface between the REA and the Pacifica Simulator is as follows.

- The REA can dispatch task requests to the simulator.
- The Simulator signals failure when a task has not successfully completed or does not have all of its preconditions satisfied.
- The REA only receives information about specific entities it requests information from (i.e., issues sensor requests for), when failures occur on executing tasks, or when a task successfully completes.
- The Simulator reports the successful completion of a task by notifying the REA of the effects of executing the task.

The Pacifica Simulator checks any preconditions which may exist for the task which it is being requested to simulate. If all of the preconditions are satisfied then the task is scheduled with a duration appropriate to the particular task. When the duration has elapsed the effects of the task are sent to the REA. If the preconditions of a task are not met, a failure is sent back to the REA with a reason as to why the failure occurred.

The REA models the types of effects which a task can be expected to yield upon successful completion. It receives two types of messages from the simulator—these being success/failure and sensor messages. The REA places all effects and sensor

information into its model of the environment so that it can base decisions upon the current situation in the environment. However, the beliefs which the REA holds about particular characteristics of entities in the environment may or may not correspond to their actual values. This is due to the fact that the EEM in the Pacifica Simulator causes changes in the world which the REA will not immediately be aware of unless it is actively monitoring for those particular changes. This provides a rich environment for studying issues related to reactive execution of plans. If a failure occurs then the REA is notified as to the reason why it occurred so that it can respond in an appropriate manner.

Interaction with the simulator can also take an interventionist form where events in the simulation can be made to occur manually. That is, events such as blown tires, the sudden loss of fuel, or a change in the location of a particular resource can be caused by a human at any time during the simulation through the Simulator Interaction Window. This is particularly useful for demonstration of specific REA capabilities.

6.3 The Pacifica Scenarios

The regional political situation for the scenario and thus the reason the NEO must be conducted, is as follows.

Civil unrest has broken out in Pacifica. Terrorists have taken over radio and television stations in Barnacle, Calypso, and the capital Delta. All modes of transportation have been disrupted including most major roads, railways, and airports. However, reports show that one airport in Pacifica, located in the capital Delta, is still under government control at this time.

The U.S. Embassy in Delta reports that 250 American Nationals are presently in the country in addition to 20 non-essential Embassy personnel. 75 Agroforestry specialists are located in the Abyssian Forest just outside of the city of Abyss, 108 World Health Organization (WHO) volunteers are located at Calypso, and 67 American University students are at Barnacle on the West coast.

The Agroforestry specialists report that they have transportation available for 25 and thus will require 50 to be transported by other means. The WHO volunteers have

transportation for 30 and the students have no transportation. Thus, the problem which must be addressed in the Pacifica-NEO is that American Nationals are located throughout Pacifica and must be extracted (airlifted) out of the country due to the civil unrest.

6.3.1 Small Scale NEO Scenario

The scenario requires that a plan be generated by a superior planning agent which will allow the REA to carry out a NEO in Pacifica. This plan contains tasks to (1) move a C5 cargo plane and two ground transports to Pacifica, (2) move the ground transports to various cities in Pacifica to pick up passengers and return them to the Point-of-Embarkation (POE), (3) move the passengers from the POE to a safe location, and (4) move the C5 and ground transports to the same location as the passengers.

The base for this operation has been selected to be in City-K, Country-X for its geographical location and onsite resources which are required to handle all aspects of the extraction operation.

Initially, the C5 cargo plane and two ground transports are located in City-K, Country-X, and the B707 passenger plane is on the ground at the airport in Delta, Pacifica. From this initial situation a plan is developed which moves the required resources from City-K, Country-X to Delta, Pacifica, transports American Nationals via ground transports from their present locations in Pacifica to Delta and finally, airlifts the Nationals to City-K, Country-X. The recovery of aircraft and ground transports airlifted to Pacifica must also be completed at the end of the operation.

6.3.2 Multiple Task NEO Scenario

The Multiple Task NEO involves the Small Scale NEO and a NEO to move medical supplies from one city to another via helicopter. This NEO plan contains tasks to (1) move a C5 cargo plane and a medical helicopter to Pacifica, (2) load the helicopter with supplies, (3) deliver those supplies to another city, and (4) return the helicopter and the C5 cargo plane to City-K, Country-X.

Initially, the C5 cargo plane and the medical helicopter are located in City-K. At some

point during the execution of the Small Scale NEO, the NEO involving the helicopter will be dispatched to the REA thus, beginning the simultaneous execution of multiple NEOs. The helicopter NEO will then cause medical supplies to be transported from Calypso to Exodus. The helicopter will return to Calypso where it will be loaded back onto the C5 and transported to City-K.

6.3.3 Scenario Assumptions

The following assumptions are made:

- Ground transports are capable of transporting up to 80 passengers.
- Fuel is readily available in each city for use by the ground transports.
- Medical Helicopters can carry supplies or 20 passengers.
- Air Cargo Transports can transport either two helicopters or two ground transports.
- Air Passenger Transports can accommodate all necessary passengers.
- Tow trucks are readily available in each city in the event that a ground transport requires such assistance.
- No terrorist attack will occur while resources are not in Pacifica.

6.4 Characterization of the Pacifica Simulator

Since we are concerned in this research with how to characterize systems it is appropriate that we characterize the simulator used to demonstrate the REA design. Here we will use the characterization of testbeds by [Hanks *et al.* 93].

Exogenous events Four types of exogenous events are present in the Pacifica Simulator. These include weather, terrorist activity, resource events, and natural disasters (see Section 6.2).

Complexity of the world The Pacifica Simulator is realistically complex for the types of problems and situations which occur during typical Non-combatant Evacuation Operations. The data is based on actual US military sources [Reece & Tate 93]. Five types of resources are modeled, along with five types of geographical entities (see Section 6.2.1).

Quality and cost of sensing and effecting Presently sensing and effecting are of high quality and no cost. However, the richness of the probabilistic model used in the Pacifica Simulator allows for noisy sensors and imperfect effectors. This characteristic can be satisfied within the design of the simulator, but is not met for the examples described in Chapter 8.

Measures of plan quality This characteristic implies that the testbed should allow problems to be posed involving partial satisfaction of desired states, forcing the agent to trade the benefits of achieving the goal against the cost of achieving it. Though this was not an area of study for the REA design, such a criterion could be met. Using the probabilistic model of the simulator one could probabilistically determine which effects of a particular task were achieved upon completion of a task. Therefore, we can say that within the design concept of the Pacifica Simulator such a characteristic could be satisfied.

Multiple Agents This characteristic is not explicitly met. That is, though the Exogenous Event Manager and the ability of a human to intervene allow for the implicit presence of other agents in the environment, the Pacifica Simulator does not allow for multiple REAs to be present in the environment.

A clean interface The REA to Pacifica Simulator interface is well-defined and strictly enforced. That is, the REA can only gather information through sensor requests, from task failures, or task completion, and the simulator does not notify the REA of any changes to the environment not specifically related to an executing task. Hence, the interface to the Pacifica Simulator is clean.

A well-defined model of time The model used in the Pacifica Simulator is one of continuous time. That is, time continues in the simulation even when the Simulator has not been specifically tasked to perform some action. This way, events

can continue to occur in the environment that can significantly change the environment between task executions.

Supporting experimentation With the probabilistic model used in the Pacifica Simulator the behavior of the simulator can be varied and controlled. However, the simulator does not presently allow for the automatic gathering of performance statistics. Therefore, this characteristic is only partially satisfied.

The Pacifica Simulator satisfies the majority of these characteristics and possesses an underlying structure that is rich enough to satisfy all of the characteristics. Therefore, it is a sufficient testbed to test the REA design. Meeting the characteristics of multiple agents and measures of plan quality addresses specific areas of research that have not been of primary concern for the research discussed in this dissertation. However, their satisfaction would allow the design of the REA to be exercised in new and interesting ways. That research will have to be considered in the future.

6.5 MAD - Prediction

In this section concerning the MAD methodology, we will describe the criteria for satisfying the characteristics of the characterization presented in Chapter 2, and discuss some expectations about the behavior we expect the design to exhibit.

6.5.1 Achieving the Three Star Rating

The basis for the REA design is the model of a rational agent as identified by the characterized capabilities. These are:

- the ability to guarantee a response in bounded time;
- the facilities to recover from execution failures and continue to operate;
- default innate behaviors which allow the agent to survive in a dynamic environment and act without having to deliberate;
- acceptance of asynchronous events to react to exogenous occurrences and changes in the environment;

- the weighing of alternative courses of action to choose actions which are most appropriate for the environmental circumstances;
- mechanisms to change the focus of attention to address more critical tasks;
- predictable behavior; and
- facilities for reasoning about time.

The success of the design to produce an agent which behaves rationally while situated in a dynamic environment must, at a minimum, be measured against these characteristics of the model. Thus, using the metric of the "star" rating, the goal is to design an agent that has a rating of "***" for each of the characteristics. This will be measured by the demonstration of the REA design in the Pacifica domain if all of the following aspects can be shown to be present for each of the characteristics.

Guaranteed Response	(1) Act with a single task directive. (2) Act without a task directive (i.e., using innate behavior). (3) Act with multiple task directives.
Failure Recovery	(1) Recover from an exogenous event which the REA can address through procedural knowledge. (2) Detect an early failure and request assistance from the superior agent. (3) Detect and recover from a failure detected by behavior mechanism. (4) Detect an exogenous event which the REA cannot address and request assistance from the superior agent. (5) Detect a precondition failure at action execution time.
Innate Behavior	(1) Act without a task directive performing some internal maintenance action. (2) Act without a task directive in response to some external stimulus.

Asynchronous Events	(1) Accept communication from the superior agent while processing a task. (2) Simultaneously processing multiple events.
Weighing Alternatives	(1) Selection of a task procedure when several are present. (2) Selection of schema when several are present. (3) Selection of a task directive to process when several are present.
Change of Focus	(1) Detection of an event by a behavior which causes the introduction of procedural knowledge to address the event. (2) Detection of a failure causing the REA to immediately address that failure. (3) Detection of a protection violation causing the REA to report such a problem to the planning agent.
Predictability	(1) Deterministic behavior in different contexts under similar circumstances.
Temporal Reasoning	(1) Ability to temporally constrain tasks before or after some event. (2) Ability to temporally order subtasks of a task.

6.5.2 Expectations

There are two primary types of factors which affect a situated rational agent. These are factors arising from the environment and factors arising from the assignment of tasks. Environmental factors are those which originate in the world in which the agent is situated, and task assignment factors are those which arise as a result of communication with a superior agent or commander.

6.5.3 Environmental

The environmental factors which must be addressed by the REA are taken from the context of the Pacifica Domain. The factors are broken down into five categories—no

unanticipated effects, side effects of an agent's own action, immediate effects, remote-effects, and beneficial effects.

No Unanticipated Effects

The simplest case of environmental issues are when no effects occur in the environment outside of the control and behavior anticipated by the REA, or the effects which occur have no direct effect on the agent.

Expectation: Execution of the plan will continue until successful completion when no unanticipated effects occur during execution.

Side Effects of an Agent's Own Action

Another simple case of environmental issues addressed are when effects occur in the environment as a result of the REA's execution of a task, yet these have no bearing on the plan being executed and the achievement of its goals.

Examples of such effects in Pacifica could be changes of resource attributes not affecting the executing task (e.g., lowering the fuel level of a resource or changing its location).

Expectation: When side effects of an agent's own action occur, execution of the plan will continue until successful completion.

Immediate Effects

A more interesting environmental issue is when exogenous effects occur which interfere with the action which is currently being executed, or when desired effects of an action just executed have not been brought to fruition and are immediately important to the next action(s) to be executed.

Examples in Pacifica of these type of effects could be a flat tire on a transport vehicle, lack of fuel in a required resource, a natural disaster closing a road or destroying a resource being used, a hostile takeover of the airport, etc.

Expectation: Immediate effects require the immediate attention of the REA since the executing task's plan cannot be continued when these types of effects are present. The result of such effects occurring is to put the REA in situations when it is potentially able to address the adverse effect, but may not be able to due to other factors (e.g., lack of commitment, lack of time, or lack of resources). These types of effects may also present problems to the REA which are beyond its capabilities and require the assistance of the planning agent. Therefore, the REA will either be able to address the effect and continue with the execution of the plan, or not be able to address the effect do to lack of capabilities, thus requiring assistance from the superior agent.

Remote Effects

Another interesting environmental issue is when effects take place in the environment which will affect either the task currently being executed, or a task which is waiting to be executed. The effects however, may not affect the execution until an action which is to be taken sometime in the future.

Such effects in Pacifica could be a road closing (e.g., by a bridge collapsing or mud slide) which is not used until the end of the task, the lack of fuel in an resource required several actions into the future, or a natural disaster occurring blocking access to a city to be visited in a future action.

Expectation: These types of effects require the REA to be able to detect them while monitoring the execution of its tasks and react to them in an appropriate manner. The immediately next action may be executable, but some future part of the plan will be in trouble. The effect may cause the entire task to fail, or may cause a delay which will violate the temporal constraints related to a particular task. Thus, when remote effects are present either the task will fail since future preconditions will not be met, or the REA will introduce new actions to the task to avoid failure. The latter case is realized when the REA detects a potential failure and requests assistance from the planning agent.

Beneficial Effects

Not all effects in the environment are to the detriment of the REA achieving its tasks. Some effects take place which are beneficial and mean that entire tasks may not have to be executed, or that portions of a plan may be skipped during execution.

Examples of these types of effects in Pacifica are a resource already being located where it is required, personnel or cargo already moved, or additional resources being available.

Expectation: Execution of the plan will continue until successful completion has been achieved.

6.5.4 Task Assignment

The task assignment factors which must be addressed by the REA are independent of the domain in which they occur. The factors fall into the categories of: comprehensible communication event, incomprehensible communication event, single task execution, multiple task execution, and inability to perform a specified task.

Comprehensible Communication Event

The simplest case of task assignment is receiving a communication event that is understood.

Expectation: The message will be passed to the Agenda Manager and be placed on the agenda for processing with the other intentions of the REA.

Incomprehensible Communication Event

Another case of communication related task assignment is when a message is received that is not understood by the REA. This occurs when a message is received which that REA has no ability to process. In other words, it is unable to understand the contents of the message.

Expectation: The message will be rejected, and a message will be dispatched to the sender notifying them that the message could not be processed.

Single Task Execution

When a message is received and placed on the agenda for processing it may involve the synthesis of a Task Directive. When no other Task Directives are being processed by the REA, then the single Task Directive will receive the full attention of the REA.

Expectation: The tasks and subtasks of the Task Directive will be executed according to the ordering and temporal constraints of that Task Directive.

Multiple Task Execution

When a message is received that calls for a Task Directive to be synthesized when one or more Task Directives already exist, then the Task Directive with the highest priority will be processed before the tasks of the other Task Directives with execution interleaved as necessary.

Expectation: Tasks selected from the agenda will be based upon the priority of the Task Directive to which the task belongs. All intentions will be executed at the priority of their corresponding Task Directive plus that of the capability that is required to process the intention.

Inability to Perform a Specified Task

When a communication event is received that specifies the REA to perform a task for which it has no knowledge of how to perform it, the REA must reject that message.

Expectation: When the REA does not possess the knowledge to carry out a particular task it has been directed to execute, it will reject the task and notify the sender that it does not have the knowledge to process the task.

6.6 Chapter Summary

We have discussed the testbed that will be used for testing the REA design. Since evaluation of the REA lies in its performance we characterized the Pacifica Simulator to determine if it satisfied the criteria for a good testbed. We saw that it did in fact meet the majority of the criteria and therefore determined that it should adequately allow us to test the REA design.

We also considered the environment and scenarios that will be used to determine if the REA design does indeed perform as claimed. We have seen that this environment is realistically complex enough to determine if the design does in fact allow the REA to behave rationally in such an environment.

In the next chapter, we will consider the failure management capabilities and features of the REA.

Chapter 7

Failure Management

Experience in planning for execution in realistic domains tells us that we cannot consistently generate plans that will succeed because of the uncertainty which is inevitably present. A planning system is not able to determine all possible interventions *a priori*, and the model of the world which it uses to base assumptions on is destined to be out of date. The situated agent which must carry out the plans generated by the planning system also has uncertainty to contend with. It is neither in total control of the environment in which it is situated, nor necessarily alone. Thus, we must be able to manage uncertainty during execution.

An execution agent must be able to comprehensively provide execution monitoring in complex and dynamic environments. One way to provide such monitoring is to: (1) monitor preconditions and essential postconditions, (2) actively monitor for protection violations, (3) actively monitor situations which are known to cause failures, (4) actively monitor for potential beneficial opportunities, (5) actively monitor the progress of an action during its execution, and (6) avoid failures or predict their potential. The research described here is to discuss how points (1), (2), (3) are addressed in the REA design and correspondingly, discuss a means of addressing (6) by detecting potential execution failures.

In this chapter we begin by discussing what causal structure is, how protection monitors are synthesized directly from plan causal structure and how protection violations can be detected early during execution using these monitors. Then in Section 7.2 we discuss the concept of active sensing that enhances the efficiency of these protection monitors.

Section 7.3 describes the behavior mechanism of the design which allows for active and passive behaviors for identifying and recovering from execution failures. Section 7.4 discusses the types of failures which are managed by the REA design, and Section 7.5 presents some research being conducted that is related to failure management.

7.1 Monitoring Protection Intervals

Many execution systems take a simplistic approach to monitoring. That is, they simply try to test preconditions and postconditions of actions to determine when execution is not going according to plan as a way of managing this uncertainty. This is a reasonable mechanism for doing so in some domains. However, as it has been shown [Doyle *et al.* 86], such an approach would not be reasonable in domains where actions take long periods of time to complete. For example, if an action N-1 completed successfully at time t1 and its effects were required by an action N-19 at t1+7 days, then the fact that some event(s) had taken place which nullified one or more of the required effects between t1 and t1+7 days would not be detected until the preconditions of the action were verified at t1+7 days.

In order to detect and resolve such problems, an execution system must actively monitor actions during their execution and subsequently monitor their necessary effect up to the point where they are required. Passive monitoring, or only checking the preconditions and postconditions, informs the execution system not to attempt to perform certain actions due to failed preconditions, or that certain actions have not produced all of their expected effects so something else must be done. Active monitoring informs the execution system on how a particular action is progressing to achieve its effects. However, active monitors as they have been defined [Sanborn & Hendler 88,Hart *et al.* 90] in the past do not address the whole monitoring picture. In addition to monitoring the progress of an action, we need active monitors to detect protection violations during protection intervals. Such violations typically manifest themselves as later failures; thus, possessing the ability to detect them provides an early warning for potential failures.

In the following sections we describe what is meant by the causal structure of a plan.

Section 7.1.2 presents a model of plan execution which is based on having casual structure information available. Section 7.1.3 discusses the process of how execution monitors are synthesized from causal structure and how they become activated, and in Section 7.1.4 we discuss what happens when a violation of a condition is detected during a protection interval.

7.1.1 Plan Causal Structure

Causal structure is a high level representation of information about a plan which states the relationship between the purposes of actions with respect to the goals or sub–goals they achieve for some later point in the plan. This information may be used by a planner to detect and correct conflicts between solutions to sub-problems when higher level plans are refined to greater levels of detail.

Various forms of causal structure representations are found in most planning systems for a variety of purposes. During plan generation its main use is for interaction detection and correction. The representations include Goal Structure (or GOST) [Tate 77,Currie & Tate 91], causal links [McAllester & Rosenblitt 91], protection intervals [Sussman 75], and plan rationale [Wilkins 84] to name a few.

During the generative planning process a causal structure table is maintained to record what facts have to be true at any point in the plan network and the possible "contributors" that can make them true. A contributor in this sense is a node in the plan network whose effects are required elsewhere in the network to satisfy a condition of another node. The planning system is able to plan without choosing one of the (possibly multiple) contributors until it is forced to by interaction of constraints. The causal structure is used to detect important interactions (ignoring unimportant side effects) and can be used to find the small number of alternative temporal constraints to be added to the plan to overcome each interaction. This "Question Answering" procedure [Tate 77] is the basis for work by Chapman on the Modal Truth Criterion [Chapman 87]. Multiple interactions arising at the same time further restrict the possible solutions and a minimal set of temporal constraints can be proposed.

We believe that this plan causal structure can be extended to represent information

which an execution agent can use to effectively monitor action execution and detect protection violations [Tate 84]. Causal structure statements represent precisely the outcome of any operation which should be monitored (i.e., protected). If lower level failures can be detected and corrected while preserving the stated higher level causal structure, the fault need not be reported to a higher level (e.g., a planning system).

7.1.2 A Model of Execution Monitoring

An execution agent is given a plan generated by a planner together with information on what the individual plan steps achieve, by what time, and for which subsequent steps (the causal structure). It must supervise the execution of actions (based on a capabilities data base which might be trivial or quite complex in nature). It should use any available monitoring capabilities to monitor the execution of each action to ensure (as far as possible) that it achieves its purpose(s).

When failures occur, recovery steps may be taken which might be of various types:

- Recovery procedures for the effector chosen (e.g., reset and repeat).
- Recovery procedures for the action type chosen (e.g., generic procedures for ensuring that an action can be successfully accomplished by passing it to a special purpose effector or skilled supervisor).
- Recovery procedures for the particular failed action (e.g., by procedural methods, etc.).

Recovery on failures can be simple or complex depending on the local intelligence of the effectors chosen, the closeness of coupling of actions in the domain, the predictability of error outcomes, etc. When a failure is found which cannot be locally recovered from within the given causal structure constraints (of required outcomes, resource usage or time limits), the execution agent must prepare a statement of the failure and changed plan circumstances to communicate back to the planner (which can then be used to suggest a plan repair).

As shown in Figure 7.1 (from [Tate 84]), an activity can be executed as soon as all the incoming causal structure requirements are satisfied (by any potential "contributor"

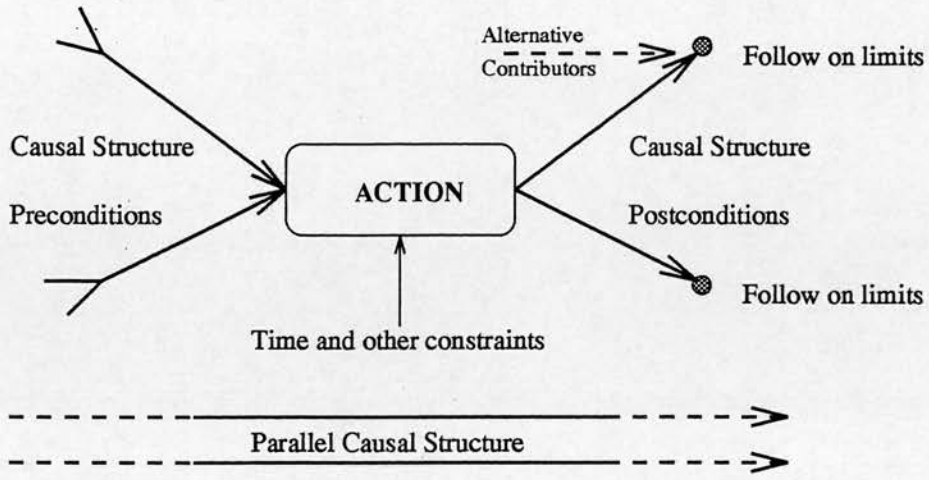


Figure 7.1: Execution from the viewpoint of a single action.

if there are several alternatives). A decision on the allocation to a particular effector must then be made. The activity and any associated constraint information is then passed to the effector.

The relevant effector executes the action and its controller must report when the activity is completed. Time-out conditions related to the time limits for the follow on actions to the causal structure outcomes are used to prevent the system hanging up on effector controller failure.

The condition monitor is triggered to check all associated causal structure outcomes of the activity. This same model of execution and condition monitoring applies when the activity involves the use of a sensor to capture information needed at some point in the plan. The causal structure outcomes in such a case my contain variables which will be bound to definite values when the condition is checked.

If failures occur, local recovery is possible (by either the effector or by using procedural methods accessible to the execution agent) within the given resource or time limits set for the follow on activities resultant on each monitored outcome. The parallel causal structure (i.e., postconditions of actions before the failed activity which are required later in the plan) provides a guide to the local recovery system on what should be preserved if the local recovery is to avoid interference with other important parts of the existing plan. Any interference with such parallel causal structure should be

reported to the execution agent as it must be re-considered by the planner to work out the actual effect on the plan.

7.1.3 Monitor Synthesis and Activation

The model described in Section 7.1.2, is the basis of the execution monitoring functionality of the REA design proposed in Chapter 4. The REA is designed to handle

```
(cstr
 ((CSTR-1  (AT GT1) DELTA (NODE-6-1) (NODE-3))
  (CSTR-2  (AT GT2) DELTA (NODE-5-1) (NODE-3))
  (CSTR-3  (RES-STATUS GT1) AVAILABLE (NODE-6-1) (NODE-3))
  (CSTR-4  (RES-STATUS GT2) AVAILABLE (NODE-5-1) (NODE-3))
  (CSTR-5  (AT C5-1) DELTA (NODE-8) (NODE-3))
  (CSTR-6  (AT GT2) ABYSS (NODE-4-2) (NODE-4-1))
  (CSTR-7  (AT GT2) DELTA (NODE-4-4) (NODE-4-3))
  (CSTR-8  (AT GT2) DELTA (NODE-8) (NODE-4-3))
  (CSTR-9  (AT GT2) DELTA (NODE-8) (NODE-4-4))
  (CSTR-10 (AT GT2) BARNACLE (NODE-5-2) (NODE-5-1))
  (CSTR-11 (AT GT2) DELTA (NODE-4-1) (NODE-5-3))
  (CSTR-12 (AT GT2) DELTA (NODE-5-4) (NODE-5-3))
  (CSTR-13 (RES-STATUS GT2) AVAILABLE (NODE-4-1) (NODE-5-4))
  (CSTR-14 (AT GT1) CALYPSO (NODE-6-2) (NODE-6-1))
  (CSTR-15 (AT GT1) DELTA (NODE-6-4) (NODE-6-3))
  (CSTR-16 (AT GT1) DELTA (NODE-8) (NODE-6-3))
  (CSTR-17 (AT GT1) DELTA (NODE-8) (NODE-6-4)))))
```

Figure 7.2: Causal structure information from a synthesize message

multiple, simultaneously executing plans and to possess the ability to monitor conditions between plan executions. This design utilizes a communication protocol called Inter-Agent Communication Language (IACL) to transmit information between the execution agent and the planner. Tasks are specified by a planning system (in the form of synthesize messages) to the REA which are then carried out using a more detailed model of the execution environment than is available to the planner. The REA executes the plans by choosing the appropriate activities to achieve the various sub-tasks within the plans, using its knowledge about the particular resources under its control. It communicates with the environment in which it is situated by executing the activities within the plans and responding to failures fed back from the environment.

Such failures may be due to the inappropriateness of a particular activity, or because the desired effect of an activity was not achieved due to an unforeseen event.

When the planner has generated a plan it intends to execute, it sends a synthesize message that contains the actions of the plan, commitment information, ordering constraints, and plan causal structure. This information is then used by the REA to synthesize a Task-Directive object which it can execute. The causal structure information contained in a synthesize message (see an example in Figure 7.2) is used by the REA to synthesize monitor objects which actively monitor for protection violations during the execution of the Task-Directive.

Each CSTR, or causal structure record, provides the execution agent with important monitoring information as follows:

(<Tag> <Pattern> <Value> <R-Node> <C-Node(s)>)

The *tag* provides a reference to the planning system for use when a failure has occurred which cannot be addressed locally by the REA. The *pattern* specifies the exact property which is to be protected for the range *C-Node(s)* to *R-Node*. The *R-Node* is the node in the plan network which *requires* the pattern to have the *Value*, and the *C-Node(s)* field specifies one or more alternative *contributors* of the value.

The causal structure record contains all the information necessary to synthesize a monitor object. The mapping of information contained in the CSTR to the monitor object is shown in Figure 7.3.

The complexity of protection monitors comes from deciding when the monitor should be active. Basically, a protection monitor is active only while the REA *intends* to execute the associated Task-Directive to which it belongs. The monitors become active immediately upon synthesis of the Task-Directive and are removed when either they expire or all actions of the Task-Directive have been executed. During the "life" of a protection monitor it could find itself in one of three states—activated, inactivated, and expired.

When a synthesize message is received by the REA and a Task-Directive object is being synthesized, any causal structure information is used to synthesize protection

Monitor-Slot	Value	CSTR-info
NAME	MONITOR-12	
TAG	CSTR-12	TAG
TASK-DIRECTIVE	#<TD-1>	
SCHEMA	#<NODE-9>	
EXPECTED-VALUE	DELTA	VALUE
KNOWN-CONTRIBUTORS	(8)	C-NODE(s)
BEING-MONITORED	(AT GT2)	PATTERN
RANGE-START	8	
RANGE-END	9	R-NODE

Figure 7.3: Monitor object created from CSTR-12

monitors and associated with that Task-Directive. All protection monitors are initially in the inactivated state when they are synthesized. For example, the causal structure information shown in Figure 7.2 is used to synthesize protection monitors for a 15 node plan giving the coverage shown in Figure 7.4. A single monitor (e.g., M12) is synthesized for each causal structure record (e.g., CSTR-12).

When the REA begins to execute any Task-Directive from its agenda the state of the monitors can change. What a protection monitor object is concerned about is when the REA's world model is updated with new information. When updates occur a set of activation-rules are applied to each protection monitor to determine if it should change its state. These rules determine whether a monitor is to be activated or has expired.

Protection monitors become activated when execution has progressed to the point where the monitor's range is valid. Once a monitor is in the activated state it remains in that state until either what is being monitored by the object does not have the value it expected (in which case it is a violation), or execution has progressed past the range-end of the monitor (in which case it has expired). Once a monitor has expired it is removed from contention and is no longer considered when the activation-rules are applied. Protection violation will be further discussed in Section 7.1.4.

An advantage of this approach to activation is that violations can be detected across Task-Directives so the planner can improve the probability that the assumptions it makes about the future will be valid by protecting them. That is, if the planner "knew" that it was, for the time being, only going to execute a portion of the plan

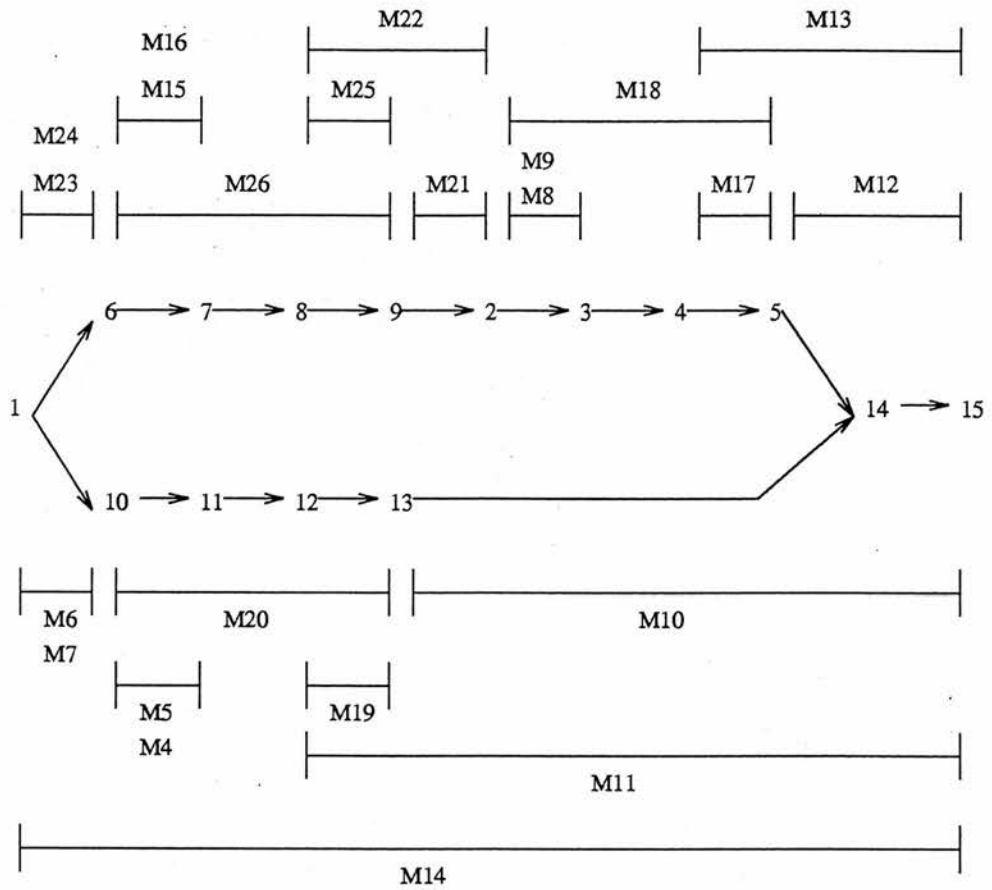
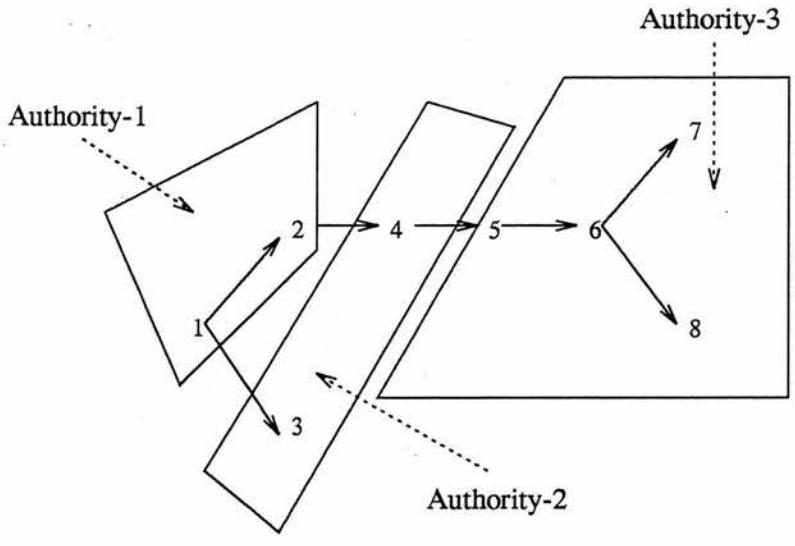


Figure 7.4: Causal Structure Coverage of a Plan by Protection Monitors

which it was working on, it could submit that plan to the REA with causal structure that would essentially protect the effects of that plan until the remaining portion of the plan could be executed. This requirement stems from the need to plan and/or execute particular "phases" of plans only to specified levels when *authority* is given to do so [Tate 93]. For example, we see in Figure 7.5, that the planner has a plan that it wishes to execute. However, in the first instance it is only given authority to execute actions N-1 and N-2. So, it sends a synthesize message to the REA with causal structure telling it that condition-1 would be introduced by action N-1 which is required by action N-2 (and some action N-3 in the future), and condition-2 would be introduced by action N-2 which would be required by some action N-4 in the future. The REA would then use the causal structure information to monitor condition-1 and condition-2 until actions N-3 and N-4 were executed.



(CSTR-1 (condition-1) (N-2 N-3) (N-1))
(CSTR-2 (condition-2) (N-4) (N-2))

Figure 7.5: Planner Authority Levels

7.1.4 Protection Violation

When updates are made to the REA's world model, the activated protection monitors are examined to determine whether a violation has occurred. When the world model is updated with new information a process within the REA is notified and informed which information changes. This process then initiates the determination of whether violations have occurred.

Each activated protection monitor that is monitoring the type of condition which changed in the world model is examined to see if the value it expects the condition to have is the same as its new value. This examination only takes place if all of the contributors of the condition have been executed (i.e., the condition should exist). If one or more contributors remain to be executed then it is likely that a premature violation has occurred, so this potential violation is ignored and the monitor remains activated. If it is the same then the monitor remains activated otherwise, if it is not the same then a violation has occurred and must be examined further.

The planner uses the causal structure to prevent plan state interactions, but the execu-

tion agent does not have the ability to prevent things from happening. Not everything in the environment is under the agent's control and some other agent might have interfered.

When a violation has occurred several considerations must be made. First, did the contributor (or last known contributor in the case of multiple alternative contributors) fail? If so, then the violation can be ignored since it was the failure to produce the condition and not any interaction in the environment. This type of failure is handled by another component of the REA. If the violation was not due to a failure then there must have been something acting in the environment which caused the violation. In this situation one of three things could be done—re-introduction, repair, or failure.

Re-introduction is the process of executing another action which will yield the same condition that was violated. However, there are several issues which must be addressed here. This can only occur if the execution agent's representation is rich enough to allow for it to perform the reasoning required to find such a candidate. Then there is the issue of what interactions the introduction of this new action could cause. It could have effects which would interact with other actions waiting to execute and cause additional violations to occur. Re-introduction allows the system to detect and correct a causal structure violation before it manifests itself as a failure in the executing plan.

The second way to address the violation is through repair initiated by the planner. In this case, the REA would communicate with the planner to inform it that the preconditions of a particular node (i.e., the range-end node) were not going to be satisfied when it is eligible to execute. This would make it the planner's responsibility to generate a repair and communicate that back to the REA so the violation was removed. The REA would then ignore any future violation detections by that particular monitor. Repair also allows the system to detect and correct a causal violation, but does so with the assistance of the planning system. It provides an early warning system so the planning system can help the REA to avert possible future execution failures.

The third measure that could be taken to address the violation would be to report total failure of the Task-Directive. Though drastic, it could save resources which could be used by other Task-Directives the REA intends to execute. Total failure is only an option when the time remaining before the need to execute the action requiring the

condition is so small as for it not to be reasonable to expect the planner to generate a repair in time.

7.2 Active Sensing

Techniques for monitoring the execution of plans provide important information when execution has not gone to plan and (as was shown in Section 7.1) in detecting potential execution failures. However, these techniques have limited effectiveness since they are dependent upon the assumption of a sufficient sensing policy. Such a policy is one where the sensing which takes place is able to detect the necessary entity value changes in the environment, is done frequently enough that timely decisions can be taken, and the cost of sensing the information is not too expensive.

The technique presented in Section 7.1 for detecting potential execution failures was designed such that the basic approach comes at no sensing cost. The protection monitors are triggered from updates to the REA's World Model and do not actually sense the environment. However, the technique is most effective when sensors are used often to keep the World Model updated. Therefore, to meet the criteria of having a sufficient sensing policy and to maximize the utility of the technique, an approach called *active sensing* has been designed into the REA.

Active sensing, like the technique for detecting potential failures, is dependent upon causal structure information from IACL *synthesize* messages. In addition, models of the sensors under the control of the REA are utilized to provide information such as maximum use frequency and to be able to determine which entities in the environment the sensor can gather information from. The active sensing approach is to actively issue sensor requests with a specified frequency for a particular sensor in order to update information in the World Model that is to be used by the protection monitors. In this way, a worse case detection guarantee can be stated such that unexpected change in the environment can be detected in a temporal window less than or equal to the period of the particular sensor required to detect those changes multiplied by the number of resources requiring the sensor. For example, if the REA possessed a sensor for gathering information about temperature and that sensor could be used with a frequency of once

every 20 minutes, the active sensing approach would allow us to guarantee that if a monitor was established for changes in this information that at worst, we would detect such changes within 20 minutes. If on the other hand, if we wanted to monitor the location of three ground transports then the guarantee would be 45 minutes (since the sensor for ground transports has a frequency of once every 45 minutes) multiplied by 3, yielding a detection guarantee of 2 hours, 15 minutes.

During the process of creating protection monitors (i.e., during Task Directive synthesis) active sensing is established on information required by the monitors via the REA's Active Behavior Manager (Section 4.3.6) according to the algorithm in Figure 7.6.

---

1. Determine if a sensing request behavior has already been established to gather the necessary information. If so, then do not establish additional active sensing for the information.
2. Otherwise, determine the resource (from the pattern information of the causal structure record) from which the information is to be gathered and determine its type.
3. Use the sensor model for the particular type of resource to determine the name of the sensor to use and its maximum use frequency.
4. Create an active behavior object that will cause a sensor request to be made of the sensor every n time units as specified by the frequency information. This active behavior will use the Sensor capability of the REA.
5. Forward the active behavior to the ABM for immediate activation.

Figure 7.6: Active Sensing Establishment Algorithm

When the Active Behavior Manager (ABM) receives an active behavior object (Section 7.3.1) it stores it in ascending order relative to the other behaviors it is managing. The order is determined according to the next time the behavior is to become active. When a new behavior is received, its next time to become active is the current time plus the period. That is, if the period is 20 minutes then the next time to become active would be now plus 20 minutes. The ABM then sleeps until the time of the activation of the first active behavior(s). When the ABM awakes it posts each activated active behavior to the Agenda Manager.

Active sensing is done over a finite period. The Sensor capability of the REA deter-

mines if a protection monitor still exists which requires an active behavior to keep the necessary information in the World Model updated. If a monitor does exist then the active behavior object is posted back to the ABM and assigned a new time to next activation. If a monitor does not exist then no further active sensing requests will be made.

For example, given the causal structure record:

(cstr-99 (temp flask-7) 97 (node-103) (node-74))

a protection monitor object would be created to monitor the temperature of flask-7 and make sure that it remained 97 (degrees Fahrenheit say) for the period from the end of node-74 to the beginning of node-103. In addition, an active behavior would be established to update the REA's World Model on a frequency determined by the model of the temperature sensor. Each time the active behavior issuing a request to the temperature sensor was activated the Sensor capability would determine if the monitor for cstr-99 still existed. While execution was on a node less than 103 for the particular Task Directive then the active behavior would be posted back to the ABM. Once execution has reached node-103 then the active behavior would not be posted back to the ABM thus, active sensing of the temperature information would end.

7.3 Behaviors

For humans, behaviors are either inherently innate or are established over time based upon experience. In order for situated agents to be considered intelligent and rational in dynamic environments they too must possess the ability to exhibit such behavior.

If we use examples of human behavior as our model, we can see the need for such behaviors. One type of behavior in humans is the act of taking a breath every 5 seconds or blinking every 3 seconds. Thus, we need a mechanism that allows particular behaviors to be exhibited on a cyclic and periodic basis (e.g., active behavior). Another type of behavior is the act of salivating when something is in our mouth, or rubbing our eye when we have an itch there. Thus, we need a mechanism that allows particular behaviors to be exhibited when certain stimuli are present or the conditions of applicability for the displaying of the behavior are present (e.g., passive behavior).

For the REA, those behaviors that correspond to inherently innate behaviors are established at initialization. Behaviors established through experience are installed at runtime via IACL *add-PS-behavior* messages (Section 5.2.2) or IACL *add-active-behavior* messages (Section 5.2.1).

Active and passive behavior objects together have three primary roles in the REA. These are:

- to periodically cause particular behaviors to be exhibited,
- to provide explicit behavior in response to internal or external stimuli, and
- to detect anticipated situations to address cognizant failures.

These active and passive behavior objects were introduced in Section 7.3 as part of the Task Behavior Language (TBL) . The purpose of the next two sections is to discuss how these TBL constructs are used in the REA and how they are used to assist in the management of execution failures.

7.3.1 Active Behaviors

Active behavior objects (Section 3.2.5) provide a means to perform tasks on a cyclic and periodic basis. Whether installed at initialization or dynamically created and added at runtime, they are managed by the Active Behavior Manager which determines when they are to become active intentions managed by the Agenda Manager (Section 4.3.3). Much of the underlying machinery for producing the periodic and cyclic nature or active behaviors, namely the ABM, has been presented previously in Sections 4.3.6 and 7.2. The purpose here is to describe what happens when an active behavior becomes activated.

The periodic exhibition of a particular active behavior is based upon its frequency. The ABM triggers an active behavior when the current time matches the behavior's timestamp (which was established when the ABM was asked to manage the behavior). Once an active behavior has become activated (i.e., triggered) it is posted to the Agenda Manager (AM) so that it can be processed.

Once triggered, an active behavior is processed according to its event information. This information informs the AM (and the Knowledge Platform) about which capability of the REA is to be used. Once the task represented by the active behavior has been performed, the capability responsible for processing the behavior posts it back to the ABM. The ABM then gives the behavior a new time-stamp that defines when the next time is that the behavior is to become activated. This is the way in which the active behaviors exhibit their specified behavior on a cyclic basis. To better understand this process, consider the following example.

Say that we have developed a REA that is capable of driving a car, and that it possesses an active behavior to check its rear-view and side mirrors every 5 seconds. For the sake of this example assume that the current time is 00:00. When the check-mirrors behavior is posted to the ABM it assigns the behavior a trigger of 00:05. At time 00:05 the ABM posts the check-mirrors behavior to the AM, which then (assuming that no other capability is currently running) sends the behavior on to the Knowledge Platform where it is to be processed by the Check-Car-Mirrors capability. The Check-Car-Mirrors capability causes the REA to dispatch tasks for viewing (i.e., sensing) the rear and side-view mirrors, then posts the check-mirrors behavior back to the ABM where it is given the time-stamp current-time plus frequency.

7.3.2 Passive Behaviors

Passive behavior objects (Section 3.2.5), like active behavior objects, can be installed at initialization or dynamically established. They are however, different from active behaviors in two respects — function and triggering.

The function of an active behavior is to perform a particular activity on a cyclic and periodic basis and not in response to other stimuli. However, a passive behavior is only triggered by internal or external stimuli as modeled in the REA's World Model. The purpose of a passive behavior is to exhibit particular behavior in response to stimuli or events. These types of behavior remain dormant while those stimuli or events are not present.

Triggering of passive behaviors is determined by the Trigger Manager (Section 4.3.5).

They are managed by the Trigger Manager (TM) as untriggered intentions (i.e., agenda entries). Each time information contained in the World Model is updated, the TM checks the triggers of all untriggered intentions to determine if that new information should cause an intention to become active (i.e., triggered). If a passive behavior is triggered then it is posted to the Agenda Manager (AM) for processing along with the other active intentions of the REA. Once the AM selects the passive behavior for processing it is processed by the capability specified by the action information of the behavior.

As an example of innate passive behavior, consider that when we are born (i.e., initialized) we possess the innate behaviors to fight-or-flee when in danger, cry when in pain, and cry when hungry. Though these are very simple behaviors we can see that similar behaviors can be exhibited for completely different reasons. For example, crying can be exhibited even though the stimuli which caused the behavior to be exhibited is completely different—breaking one's arm versus wanting dinner. The mechanism within the REA for specifying passive behaviors allows us to exhibit this same behavior. To do this, we specify two behaviors each with different triggering conditions, but with the same action to be taken upon triggering.

For passive behaviors established over time, consider the act of tying your shoes. Once our parents got fed up with having to tie our shoes for us, they taught us to tie our own shoes. Thus, we acquired the capability to tie our shoes. However, though we had the capability, we still had to be told (i.e., tasked) to do so. As we became older we learned to identify when our shoes were untied and used our capability of tying shoes without having to be told. Therefore, the behavior became part of our overall set of behaviors that we are able to exhibit. As we discussed in Section 5.2.2, the specification of passive behaviors with new knowledge and capabilities allows the REA to adapt its behavior. Thus, the REA is able to possess inherently innate behaviors, as well as establish them over time.

7.4 Failure Management

The REA has been designed to manage the following types of execution failure:

- Non-explicit failure,
- Explicit anticipated failure, and
- Explicit unanticipated failure.

Non-explicit failure is the failure to achieve the desired effects of a task due to lack of knowledge. Explicit anticipated failures are those failures that are known to be possible in the environment for which the REA has specific repair strategies. The third type of failure occurs when an unanticipated event causes a task to fail and a strategy to repair such a failure is beyond the knowledge and capabilities of the REA. All that can be hoped for in this case is the graceful degradation from the failure with continued normal operation.

This section describes how the design of the REA allows it to address each of these execution failure types.

7.4.1 Failure to Achieve Effects (non-explicit failure)

The failure to achieve the desired effects of a task is due to the fact that the REA possesses inaccurate domain knowledge about the tasks it is to execute in the environment, or that something beyond the control of the REA has occurred during the execution of a task. Either there are no procedures eligible to execute in the current context (i.e., present state of the environment), the REA attempts to execute tasks based upon stale information in the World Model and those tasks failure since their preconditions are not actually met, or a task executes without having achieved its effects for one reason or another.

When such failures occur, the REA employs several tactics in an attempt to get execution back on track. These tactics are:

- Repeat the task,
- Select an alternative procedure, or
- Select an alternative task to bring about the same effects.

Repeating a task in the face of failure leads to robust execution, but as Firby [Firby 89] points out it also leads to potential problems since the system may repeatedly attempt to execute a single task leading to a futile loop. The approach in the REA is to only allow tasks to be repeated if they are explicitly expressed as repeatable using the TBL construct repeat-while (Section 3.2.2). In the RAP system [Firby 89] a simple means of addressing the futile loop problems was used, and no additional insight is provided in the REA design presented in this dissertation. Neither approach is satisfactory and a true solution has clearly not emerged from the reactive agent community as of yet. What is clear is that an intelligent agent must be able to repeat tasks due to the uncertainty in modeling dynamic environments and by the very nature of such environments tasks are not always going to execute the first time they are tried.

If a task cannot be repeated then the REA determines if there is an alternative procedure for the task that is applicable in the current context. Given that such a procedure is available and eligible to be executed in the current context it is executed. Otherwise, the REA must determine if there is another task available which, if executed, could bring about the same effects as the one which has failed. If such a task is found and its preconditions are satisfied in the current context then it is executed.

When the REA has exhausted these tactics without success it notifies the planning agent via an IACL *execution-failure* message (Section 5.2.5) and stops the execution of the Task Directive to which the failed task belongs. By sending this message the REA passes responsibility for the continued execution of the Task Directive to the planning agent. This allows the REA to execute any other Task Directives that may be awaiting execution.

The theory behind this approach is that the REA has done everything possible to execute the Task Directive. Since it was unable to successfully achieve the desired result it is up to the planning agent to replan, create a new plan to achieve the result, provide additional necessary knowledge and/or capabilities, or some combination thereof, to assist the REA. Once the planning agent has developed a solution it can dispatch that to the REA via IACL messages, and the REA treats it as a new Task Directive.

7.4.2 Anticipated Failures (explicit failure)

When we design control systems, we do so for particular environments where we know some of the types of failures that are likely to occur. It is highly unlikely, for all but trivial environments, that we would be able to anticipate all of the failures known to occur, but we can anticipate some. For those which we can anticipate we must provide a means for our system to address those specific cases so the system will perform with some amount of robustness.

To meet this need the REA has been designed with a mechanism that allows it to apply general or specialist knowledge to address anticipated or known failures that may occur in the environment that it is situated. General failure knowledge is knowledge about anticipated failures that are part of the Failure capability when the REA is initiated. However, since the Inter-Agent Communication Language allows knowledge to be dynamically incorporated into the REA's overall base of knowledge, we should look to see if that knowledge applies to failures the REA might encounter. Therefore, the Failure capability determines if there is any such specialist knowledge that is available before attempting to apply its general knowledge.

When a passive behavior is specified it includes information as to the problem(s) it handles. This information along with the action information is stored in a data structure that maps these problems to the actions (i.e., capabilities) that are used to address them. This mapping forms the specialist knowledge that the Failure capability considers when addressing a failure.

It may seem redundant to have a passive behavior that is specified to address such problems and a capability which tries to apply this same knowledge in failure situations. However, though the passive behavior is used to address the same problem it would typically not be triggered by under the same conditions. When a passive behavior is specified it contains trigger information which states the specific conditions under which it is to become activated. These conditions are based upon information in the World Model. When a failure occurs a type of failure is returned which is not asserted into the World Model. Thus, the passive behavior would not trigger. The advantage of this approach is that the Failure capability is able to draw upon all knowledge available

in the REA to address failures even though that knowledge may not have originally been intended for that purpose.

When an explicit failure is reported from the environment the REA uses its Failure capability to determine how to address the failure (if possible). What is reported to the REA is a type of failure or a "reason" for the failure. This information is first used to determine if the REA possesses any specialist knowledge to address the failure type. If specialist knowledge for the particular failure type is available then it is used to handle the failure. Otherwise, the failure type information is used to determine if the REA possesses any general knowledge to address the failure.

Chapter 8 will describe the process of handling failures in more detail by giving specific examples with the Pacifica Simulator.

7.4.3 Failure beyond Knowledge and Capabilities

When a failure occurs which was unanticipated, we expect the REA to report such a failure to the planning agent, stop executing the Task Directive involved in the failure, and continue its normal operation. A failure that occurs for which there is no specialist knowledge or general knowledge is deemed beyond the knowledge and capabilities of the REA.

The REA reports the failure to the planning agent via the IACL *execution-failure* message. It attempts to determine the resource involved in the failure in order to:

- identify the failed task,
- identify the failed Task Directive, and
- identify the affected causal structure records.

The REA does this assuming that such information will be useful to the planning agent in replanning. Once such information is gathered and sent to the planning agent, the REA attempts to continue normal operation by executing any other waiting Task Directives or continues the concurrent execution of its other intentions.

What makes failure management difficult for the REA is the fact that it is capable

of executing multiple Task Directives simultaneously and primitive tasks are executed asynchronously. For this reason, the failure knowledge applied is procedural in nature and dependent upon the type of failure. The basic approach for a particular failure type is:

1. determine the affected resource,
2. determine which task(s) are appropriate to address the repair,
3. determine which of the appropriate tasks are eligible to execute in the current context,
4. select a task to repair the failure,
5. select a procedure of the repair task, and then
6. execute the procedure.

If the REA cannot determine the affected resource then the other steps cannot be carried out. In such a case, the REA notifies the planning agent that it has no knowledge to address the failure.

7.4.4 Failure Management Examples

In order to get a feel for how the failure management capabilities of the REA actually come to bear, we consider two situations from the Small Scale NEO Scenario in which the capabilities are used. The first is a precondition failure that can be addressed by the REA's knowledge, and the second, an exogenous event that causes a failure which the REA must address.

Examples

Once the planner has developed the Small Scale NEO plan it wishes the REA to carry out on its behalf, it packages that plan in the form of an IACL *synthesize* message. The message contains information such as ordering constraints on the actions, resources

required, the teleology related to the plan, temporal constraints, and commitment to its achievement. The *synthesize* message is then communicated to the REA.

The Small Scale NEO plan contains the following tasks:

1. (fly-transport delta city-k)
2. (drive gt2 barnacle delta)
3. (load gt2 barnacle)
4. (drive gt2 delta barnacle)
5. (unload gt2 delta)
6. (drive gt1 calypso delta)
7. (load gt1 calypso)
8. (drive gt1 delta calypso)
9. (unload gt1 delta)
10. (drive gt2 abyss delta)
11. (load gt2 abyss)
12. (drive gt2 delta abyss)
13. (unload gt2 delta)
14. (fly-passengers city-k delta)
15. (fly-transport city-k delta)

However, these are not necessarily directly executable by the REA. The REA further decomposes these tasks to lower level tasks which themselves may have to be decomposed further. Once a task has been decomposed to a primitive action (which can be executed in the environment) it is eligible to be dispatched.

Since no EEM event can affect a resource while outside of Pacifica, the first task of interest is the unloading of the c5 cargo plane in Delta. The `(fly-transport delta city-k)` task is decomposed by the REA to the following tasks.

1. (load c5 city-k)
2. (fly-to-dest c5 city-k delta)
3. (unload c5 delta)

```
4:29:13 Completed: (Unload-Plane c5 at delta)
...
4:31:30 (E) Gt2 now has 0 gallons of fuel.
4:31:57 Started: (Drive gt1 from delta
                  to calypso on road-cd at 54)
4:32:46 FAILURE: (Drive Gt2 from Delta
                  to Barnacle on Road-Bd at 37)
4:32:46 (I) Gt2 traveled 0 miles of 85 along Road-Bd
            before the FAILURE: No-Fuel-Gt!
...
4:34:12 Started: (Refuel gt2)
4:35:24 (E) Volcanic activity...Road-BD is now closed.
4:50:17 Completed: (Refuel gt2)
4:50:17 (I) gt2 now has 55 gallons of fuel.
...
4:54:24 Started: (Drive gt2 from delta
                  to barnacle on road-cd at 54)
```

Figure 7.7: Simulation History Snapshot: GT2 No Fuel Failure from Small Scale NEO Scenario

4. (refuel c5)

During the unloading of the c5 (since the gt1, gt2, and c5 resources are now in Pacifica) the EEM can generate events which can cause failures. Take for instance, the case where the EEM sets the fuel level of the gt2 resource to zero just before it is to travel down Road-BD (i.e., the road between Barnacle and Delta). Upon being informed of the fact that the c5 cargo plane has been unloaded the REA examines the Task Directive (TD) for the next task to carry out. It finds that tasks 2 and 6 are both eligible to be executed. It decomposes each task and dispatches tasks to the Pacifica Simulator to tell it to drive the gt2 resource to Barnacle on Road-BD and drive the gt1 resource to Calypso on Road-CD (see Figure 7.7). The gt1 resource sets out along Road-CD however, the simulator informs the REA that it cannot do the task involving the gt2 resource by issuing a failure of type "no-fuel-gt." The REA immediately checks for any specialist capability it may have to address the no-fuel-gt failure. It finds none, so it checks for any general knowledge it may have to address this type of failure. It finds that it is able to address the failure by issuing a task of `(refuel gt2)` which it promptly dispatches to the Pacifica Simulator. Once the refuel task has completed,

the simulator notifies the REA that the gt2 now has 55 gallons of fuel (which is a full tank). The REA then locates the original task to have the gt2 resource drive down Road-BD and re–issues that task to the simulator. The simulator, upon receiving the "drive" message, checks the preconditions for driving. It finds that the preconditions are satisfied however, Road-BD cannot be used since volcanic activity has blocked access to it, so the simulator informs the REA of this fact[1]. This information makes the "drive" task be reconsidered by the REA and it calculates a new route for the gt2 resource to use to get to Barnacle. It chooses the route Road-CD to Road-BC, and issues "drive" task to the simulator telling it to use Road-CD instead of Road-BD.

Another example of how the REA handles unexpected events is in addressing a blown tire on a ground transport resource (see Figure 7.8). Here the c5 has just been unloaded and the gt1 and gt2 ground transports have started traveling along Road-CD and Road-BD respectively. At time 3:18:46 during the simulation a blown tire event is induced to happen on the gt1 ground transport. This causes the failure "nail-in-tire" to be signal by the Pacifica Simulator to the REA. The REA immediately checks for any specialist capability it may have to address the nail-in-tire failure. It finds none, so it checks for any general knowledge it may have to address this type of failure. It finds that by using its general knowledge it is able to address the failure with its capability `fix-gt-tire-on-road`. This capability then causes the task `(gt-tire-tow-and-repair gt1)` to be dispatched to the Pacifica Simulator. This task causes the simulator to drive a tow truck out from the city nearest to the broken down gt1 (in this case, Delta), tow gt1 back to that city, and repair the tire. Once the gt-tire-tow-and-repair task has completed, the simulator notifies the REA that the tire status of the gt1 resource is "okay." The REA then locates the original task to have the gt1 resource drive down Road-CD and re–issues that task to the simulator. The simulator, upon receiving the "drive" message, checks the pre–conditions for driving. It finds that the pre–conditions are satisfied and sets the gt1 ground transport off along Road-CD again.

---

[1] If a road which the REA specifies for a drive task is not accessible the Pacifica Simulator does not send a failure message to the REA. Instead, it sends this information to the REA as an effect of the REA having issued the drive task. The reason for this is to demonstrate the looping ability in the REA. It could just as well have been implemented as a failure which the REA had to address.

```
2:58:59 Completed: (Unload-Plane c5 at delta)
...
3:01:42 Started: (Drive gt1 from delta
                  to calypso on Road-cd at 54)
3:02:31 Started: (Drive gt2 from delta
                  to barnacle on Road-cd at 54)
...
3:18:46 FAILURE: (Drive Gt1 from Delta
                  to Calypso on Road-Cd at 54)
3:18:46 (I) Gt1 traveled 15 miles of 50 along Road-Cd
            before the FAILURE: Nail-In-Tire!
3:20:23 Started: (GT-Tire-Tow-and-Repair of gt1 to delta)
3:51:05 Completed: (GT-Tire-Tow-and-Repair of gt1 to delta)
3:51:05 (I) Tire Status of gt1 is now Okay.
3:52:44 Started: (Drive gt1 from delta
                  to calypso on road-cd at 54)
...
```

Figure 7.8: Simulation History Snapshot: GT1 Nail in Tire Failure from Small Scale NEO Scenario

7.5 Related Work

There have been many interesting avenues of research pursued in the last decade related to execution failure management. However, these are typically from the perspective of a planning system or a situated agent with planning capabilities. Nonetheless, this research is still applicable to the approach of failure management in the REA where failures are beyond its control are passed to a planning agent for assistance. This section briefly presents some of the related research[2].

[Howe & Cohen 91] present a model of failure recovery that considers the cost of various methods for recovering from execution failures. They use a mapping of domain-independent recovery methods onto failure situations known to be present in the environment that their agent is situated. Their results, though limited to a single environment and described for an agent with replanning capabilities, are promising. The same sort of mechanism could be used by the REA allowing it to have multiple gen-

[2] The interested reader should also see [Doyle *et al.* 86,Ambros-Ingerson & Steel 88,Hart *et al.* 90,Simmons 90].

eral and specialist methods of handling execution failure. [Burnell 94] uses a similar approach, selecting a recovery strategy based upon the type of failure, criticality of the failure, availability of resources and knowledge involved in a plan failure. Her approach however, uses probabilistic models, represented as belief networks that determine the likelihood that an error resides on a particular execution path.

[Wilkins 85b] addresses the problem of transforming a plan when an execution failure occurs, retaining as much of the original plan as possible while still accomplishing the original goal. He describes eight replanning actions that are domain-independent, but also prove useful as a basis for domain-specific error recovery methods. The disadvantage of this approach in applying it to the REA design is that it takes advantage of the underlying structure of the plan maintained by the planner. For an execution system like the REA, such structure is not usually available, nor are the mechanisms necessary to manipulate such structure. However, some of this structure is in the form of causal information which we have seen can be used by the REA (Section 7.1). This information allows an execution agent to detect (and possibly correct) potential failures before they manifest themselves as actual failures in a executing plan. It also provides a means for developing an early warning system so a planning system can assist the execution agent to avert execution failures by suggesting repairs.

For an execution system it is one thing to know where a plan has failed, and another to be able to repair the plan from that point. The work presented here has shown how to detect failures and how to detect potential failures in order to give a planning system more time to develop a solution. The next phase in the development of a comprehensive recovery mechanism for a design similar to that of the REA — where planning and execution are concurrent — is to develop mechanisms for planning agents that can use the information from execution failures detected by the REA. Some interesting research on the use of causal structure information in plan reuse and modification is being done to address such issues [Tate 84,Kambhampati 90]. Kambhampati uses a *validation structure* to represent the internal dependencies of a plan (i.e., causal structure) and then uses that structure to help in modifying plans to suit new situations. Work on insertion of recovery plans by [Musliner *et al.* 91] supports the "assistant" approach advocated here, but they are careful to point out that such an approach is valid only

when applied to reversible execution errors.

7.6 Chapter Summary

The value of using causal structure information in planning systems has been widely recognized. However, its utility in execution systems has not received much attention. The benefits of providing such information to execution systems are realized during deliberation and while reacting to change. The planning system is able to reduce the uncertainty of the information in its model of the world by tasking an execution system to monitor conditions it expects to be valid in the future.

We have seen how the execution system is able to avert potential failures by identifying them sooner thus, giving it (and the planning system) more time to make repairs. In addition, the concept of active sensing was introduced that allows a guarantee to be stated for a worse case detection on a particular value that is being monitored. We have also discussed the types of failures the design allows the REA to address.

In the next chapter, we consider execution of examples of the REA design in the Pacifica Simulator to determine if the design meets the criteria established in Section 6.5.1.

Chapter 8

Execution Examples

The goal of the research described here has not been to produce a complete execution system, but rather show how various mechanisms could be developed and integrated in a particular architecture that would allow for competent, rational behavior in complex and dynamic environments. A great deal of work still remains before a system such as the REA could actually be used to control real–world entities in such environments, not to mention there are many issues which still need to be addressed. Nonetheless, we can discuss the validity of the mechanisms employed in the design to provide our characterized behaviors, and discuss the additional concepts of dynamic adaptation through communication, causal structure based protection monitors for predicting execution failure, active sensing, and active and passive behaviors.

In this chapter we use the complex and dynamic environment provided by the Pacifica Simulator (Chapter 6) to determine if the REA design successfully meets the criteria for each of the characterized features according to the '***' rating system.

We begin with a detailed example of execution to familiarize ourselves with the output format of the REA. Then in Section 8.2 we assess the performance of the design by determining whether it satisfies the criteria of the characterization set forth in Section 6.5.1. Necessitated by a desire for a clear understanding of how each of the characteristics are satisfied, the detail required for some of the examples is extensive. As a result this chapter is quite verbose.

8.1 Example

In order to get an understanding of the inner workings of the REA, and the output format of that reasoning, we consider the execution of the task fly-plane-to-dest. We join the execution trace where the REA has already synthesized a Task Directive for the Small Scale NEO Scenario, and is currently trying to execute the task fly-transport. The fly-transport task is composed of two network procedures—fly-transport-1 and fly-transport-2. The REA having chosen fly-transport-1 has just completed the load task and is now attempting to fly the C5-A Galaxy cargo plane (c5-1) from City-K, Country-X to Delta on the island of Pacifica.

The locus of control for a particular capability of the REA is identified by lines which contain ":KS-<capability-name>0 ***..." These lines bound a complete cycle of processing for a particular capability. Within these bounds we can see the Agenda Entry (AE) or intention being processed, the reasoning by the capability to determine what to do, and the posting of tasks for processing by other capabilities to the agenda. Also included in the trace are snapshots of the agenda so we can see what intentions the REA has throughout the execution of the fly-plane-to-dest task. The snapshots of the agenda are bounded by hash marks and are shown at the beginning of each cycle (unless otherwise noted). The AE numbers and the cycle numbers are not related, but they appear to be in this example due to unfortunate coincidence.

We join the execution in cycle 33, where the Execute capability is selecting the next task to execute from the network task fly-transport-1.

```
####################################################
                Agenda (cycle:   33)

<AE-8  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>

-------------- Untriggered Entries --------------

No Untriggered entries...

####################################################

:KS-EXECUTE0 **********************************
<AE-8/0:
```

```
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT-GROUP
                ((OR (LOAD-STATUS (QUOTE C5-1) (QUOTE LOADED))
                     (AND (NATIONALS-AT (QUOTE CITY-K) (QUOTE NO))
                          (NOT (O-TYPE (QUOTE CITY-K) (QUOTE BASE)))
                          (CARGO (QUOTE CITY-K) (QUOTE NO))))))
     BODY: (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-TRANSPORT-1(NETWORK) to Execute...
 FLY-TRANSPORT-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~01:47:32...Considering task FLY-TRANSPORT-1 with status P
              of (FLY-TRANSPORT C5-1 DELTA CITY-K)
 0~01:47:32...Network of FLY-TRANSPORT-1 is:
              (#<SCHEMA NODE-17.0 FLY-PLANE-TO-DEST>
               #<SCHEMA NODE-18.0 ACT-SENSOR>
               #<SCHEMA NODE-19.0 UNLOAD>
               #<SCHEMA NODE-20.0 REFUEL-PLANE>)
 0~01:47:33...Eligible procedures are:
              (#<PROCEDURE FLY-PLANE-TO-DEST-1>)
<== Posting #<PROCEDURE FLY-PLANE-TO-DEST-1> for :EXECUTE ==>
 0~01:47:33...Creating task FLY-PLANE-TO-DEST-1 of FLY-TRANSPORT-1
<== Posting #<PROCEDURE FLY-TRANSPORT-1> for :EXECUTE ==>
<== Trigger: ((AND (AT (QUOTE C5-1) (QUOTE DELTA))
                   (PARKED-AT-GATE (QUOTE C5-1) (QUOTE YES)))) ==>
 0~01:47:33...Suspending task FLY-TRANSPORT-1
              of (FLY-TRANSPORT C5-1 DELTA CITY-K)
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******
```

In cycle 33, examining AE-8 we see that it was waiting for the C5-1 resource to have a load–status of loaded or the conjunction of some other conditions. Here AE-8 was triggered by the completion of the load-2 procedure of fly-transport-1 when it reported that the C5-1 resource was loaded. The Execute capability then selects the next task to be executed from the network according to temporal and ordering constraints on the fly-transport-1 procedure. It selects the fly-plane-to-dest task, and a procedure to carry out that task is selected (i.e., fly-plane to-dest-1). The procedure is then posted to the agenda for processing by the Execute capability. Next, the execution of the fly-transport-1 procedure is suspended until the completion of the fly-plane-to-dest-1 procedure. This is done by posting fly-transport-1 to the agenda with a trigger that states the context in which it is eligible to be processed again. We see the results of this activity in the agenda snapshot for cycle 34.

```
##################################################
              Agenda (cycle:   34)

<AE-26  :  95 (EXECUTE #<PROCEDURE LOAD-2>)>
<AE-35  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>

-------------- Untriggered Entries --------------
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-EXECUTEO **********************************
<AE-26/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT-GROUP
                ((OR (LOAD-STATUS (QUOTE C5-1) (QUOTE LOADED))
                     (AND (NATIONALS-AT (QUOTE CITY-K) (QUOTE NO))
                          (NOT (O-TYPE (QUOTE CITY-K) (QUOTE BASE)))
                          (CARGO (QUOTE CITY-K) (QUOTE NO))))))
     BODY: (EXECUTE #<PROCEDURE LOAD-2>)
     CONTROLLER-INFO: (C5-1)>

 Receiving LOAD-2(NETWORK) to Execute...
 LOAD-2 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~01:47:33...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~01:47:33...Network of LOAD-2 is empty.
 0~01:47:33...LOAD-2 has completed.
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

In cycle 34, the load-2 procedure is processed by the Execute capability. It is determined that there are no more subtasks of load-2 to be executed and that its effects have been achieved[1]. No further processing of load-2 takes place and the Knowledge Platform notifies the Agenda Manager that it is ready to process the next intention. Examining the agenda snapshot for cycle 35 we see that the next intention to be processed is the fly-plane-to-dest-1 procedure.

```
##################################################
              Agenda (cycle:   35)

<AE-35  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
```

[1] The achievement of effects for a task is determined by querying the REA's World Model.

```
--------------- Untriggered Entries ---------------
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-EXECUTEO **********************************
<AE-35/0:
     PRIORITY: 95
     TRIGGER: T
     BODY: (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-PLANE-TO-DEST-1(NETWORK) to Execute...
 FLY-PLANE-TO-DEST-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status R.
 0~01:47:33...Considering task FLY-PLANE-TO-DEST-1 with status R of
              FLY-TRANSPORT-1
              FLY-PLANE-TO-DEST-1 is a NETWORK task.
              The network is ((TAXI C5-1) (GET-CLEARANCE C5-1)
                              (LIFTOFF-FLY C5-1 CITY-K DELTA)
                              (LAND C5-1 DELTA))
<== Posting #<PROCEDURE TAXI-1> for :EXECUTE ==>
 0~01:47:34...Creating task TAXI-1 of FLY-PLANE-TO-DEST-1
<== Posting #<PROCEDURE FLY-PLANE-TO-DEST-1> for :EXECUTE ==>
<== Trigger: ((RES-STATUS (QUOTE C5-1) (QUOTE RUNWAY-HOLDING))) ==>
 0~01:47:34...Suspending task FLY-PLANE-TO-DEST-1
              of FLY-TRANSPORT-1
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

Here we see that at day 0, 1 hour and 47 minutes, the Execute capability receives the network procedure fly-plane-to-dest-1 that has an execution status of ready (i.e., R). The network for this procedure is composed of four tasks—taxi, get-clearance, liftoff-fly, and land. Of those, only the taxi task is cleared to execute according to temporal and ordering constraints. The procedure taxi-1 is selected from the taxi task and is posted to the agenda for processing by the Execute capability. The execution of the fly-plane-to-dest-1 procedure is then suspended until the completion of the taxi-1 procedure (i.e., until the effects of the taxi-1 procedure have been realized in the environment).

```
##################################################
               Agenda (cycle:   36)
```

```
<AE-37  : 95  (EXECUTE #<PROCEDURE TAXI-1>)>

--------------- Untriggered Entries ---------------
<AE-38  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-EXECUTEO **********************************
<AE-37/0:
     PRIORITY: 95
     TRIGGER: T
     BODY: (EXECUTE #<PROCEDURE TAXI-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving TAXI-1(PRIMITIVE) to Execute...
 TAXI-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status R.
 0~01:47:34...Considering task TAXI-1 with status R of
              FLY-PLANE-TO-DEST-1
              TAXI-1 is a PRIMITIVE task.
<== Posting #<PROCEDURE TAXI-1> for :RESOURCE ==>
 0~01:47:34...Dispatching task TAXI-1 of FLY-PLANE-TO-DEST-1
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

The Execute capability receives the taxi-1 procedure, determines that it is a primitive, and checks that its effects are not already satisfied according to the REA's World Model. The taxi-1 procedure is then posted to the agenda for processing in cycle 37 by the Resource capability. The Resource capability provides a means for the REA to perform some simple resource reasoning by checking that it has not dispatched a task requiring, in this case, the c5-1 resource. That is, it attempts to avoid dispatching multiple tasks for a single resource. The processing by the Resource capability is not explicitly shown in the execution trace since it is not fully implemented. The Resource capability, once determining that it is okay to dispatch a task, posts the task to the back to the agenda for processing by the Dispatch capability. This is the posting message shown between the agenda snapshots for cycles 37 and 38. For the remainder of this example we shall just say that a particular task has been dispatched, and not refer to the processing by the Resource capability.

```
##################################################
              Agenda (cycle:   37)


<AE-39  : 95 (RESOURCE #<PROCEDURE TAXI-1>)>

-------------- Untriggered Entries --------------
<AE-38  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

<== Posting #<PROCEDURE TAXI-1> for :DISPATCH ==>

##################################################
               Agenda (cycle:   38)


<AE-40  :  95 (DISPATCH #<PROCEDURE TAXI-1>)>

-------------- Untriggered Entries --------------
<AE-38  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-DISPATCHO **********************************
 0~01:47:29...(TAXI C5-1) dispatched.
:KS-DISPATCHO ****** END OF KS-DISPATCH ******
```

In cycle 38, at time 1 hour 47 minutes, the taxi-1 procedure is dispatched by the Dispatch capability thus, informing the c5-1 resource that is to taxi to the end of the runway. When the c5-1 resource has done that it notifies the REA. In cycle 39 (not shown), the REA receives the report from the environment (in this case the Pacifica Simulator) that the resource-status of the c5-1 resource is runway–holding which triggers the processing of the fly-plane-to-dest-1 procedure by the Execute capability in cycle 40. For the remainder of this example, when agenda cycles are not explicitly described it is this sort of updating of the REA's World Model from reports from the environment which is taking place.

```
...World Model updated from sensor reports in cycle 39...
```

```
##################################################
               Agenda (cycle:   40)

<AE-38   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>


-------------- Untriggered Entries --------------

<AE-36   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-EXECUTE0 **********************************
<AE-38/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT
                 ((RES-STATUS (QUOTE C5-1) (QUOTE RUNWAY-HOLDING))))
     BODY: (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-PLANE-TO-DEST-1(NETWORK) to Execute...
 FLY-PLANE-TO-DEST-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~01:52:40...Considering task FLY-PLANE-TO-DEST-1 with status P of
              FLY-TRANSPORT-1
 0~01:52:40...Network of FLY-PLANE-TO-DEST-1 is:
              (#<SCHEMA NODE-34.0 GET-CLEARANCE>
               #<SCHEMA NODE-35.0 LIFTOFF-FLY>
               #<SCHEMA NODE-36.0 LAND>)
 0~01:52:41...Eligible procedures are: (#<PROCEDURE GET-CLEARANCE-1>)
<== Posting #<PROCEDURE GET-CLEARANCE-1> for :EXECUTE ==>
 0~01:52:41...Creating task GET-CLEARANCE-1 of FLY-PLANE-TO-DEST-1
<== Posting #<PROCEDURE FLY-PLANE-TO-DEST-1> for :EXECUTE ==>
<== Trigger: ((CLEARANCE (QUOTE C5-1) (QUOTE YES))) ==>
 0~01:52:41...Suspending task FLY-PLANE-TO-DEST-1
              of FLY-TRANSPORT-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******
```

In cycle 40, the Execute capability continues to execute fly-plane-to-dest-1 procedure. It selects the get-clearance task as the next task to be executed, and selects the get-clearance-1 procedure of that task. It posts the get-clearance-1 procedure to the agenda and again suspends the execution of the fly-plane-to-dest-1 procedure. In cycle 41, the Execute capability processes the get-clearance-1 procedure.

```
##################################################
                Agenda (cycle:   41)


<AE-42  :  95 (EXECUTE #<PROCEDURE GET-CLEARANCE-1>)>

--------------- Untriggered Entries ---------------
<AE-43  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-EXECUTEO **********************************
<AE-42/0:
     PRIORITY: 95
     TRIGGER: T
     BODY: (EXECUTE #<PROCEDURE GET-CLEARANCE-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving GET-CLEARANCE-1(PRIMITIVE) to Execute...
 GET-CLEARANCE-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status R.
 0~01:52:42...Considering task GET-CLEARANCE-1 with status R of
              FLY-PLANE-TO-DEST-1
              GET-CLEARANCE-1 is a PRIMITIVE task.
<== Posting #<PROCEDURE GET-CLEARANCE-1> for :RESOURCE ==>
 0~01:52:42...Dispatching task GET-CLEARANCE-1 of FLY-PLANE-TO-DEST-1
:KS-EXECUTEO ****** END OF KS-EXECUTE ******

...The Resource capability was processed in cycle 42...

##################################################
                Agenda (cycle:   43)


<AE-44  :  95 (DISPATCH #<PROCEDURE GET-CLEARANCE-1>)>

--------------- Untriggered Entries ---------------
<AE-43  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-DISPATCHO **********************************
 0~01:52:43...(TOWER-CLEARANCE C5-1) dispatched.
:KS-DISPATCHO ****** END OF KS-DISPATCH ******
```

One thing that may be confusing in this part of the execution is the dispatching of the get-clearance-1 task as tower-clearance. This is in fact, the responsibility of the Dispatch capability. It takes a task and determines what it needs to communicate to the hardware, agent, or in this case, simulator in order to get that task to be executed. Here the Pacifica Simulator would not understand get-clearance, but it does have a task tower-clearance that achieves the same effects.

In cycle 45, the REA has been notified (in cycle 44) that the c5-1 has received clearance, therefore processing by the Execute capability on the fly-plane-to-dest-1 procedure is triggered.

```
...World Model updated from sensor reports in cycle 44...

####################################################
                Agenda (cycle:   45)

<AE-43   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>


--------------- Untriggered Entries ---------------

<AE-36   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

####################################################

:KS-EXECUTEO **********************************
<AE-43/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT
                ((CLEARANCE (QUOTE C5-1) (QUOTE YES))))
     BODY: (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-PLANE-TO-DEST-1(NETWORK) to Execute...
 FLY-PLANE-TO-DEST-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~01:54:42...Considering task FLY-PLANE-TO-DEST-1 with status P of
              FLY-TRANSPORT-1
 0~01:54:42...Network of FLY-PLANE-TO-DEST-1 is:
              (#<SCHEMA NODE-35.0 LIFTOFF-FLY>
               #<SCHEMA NODE-36.0 LAND>)
 0~01:54:42...Eligible procedures are: (#<PROCEDURE LIFTOFF-FLY-1>)
<== Posting #<PROCEDURE LIFTOFF-FLY-1> for :EXECUTE ==>
```

```
 0~01:54:42...Creating task LIFTOFF-FLY-1 of FLY-PLANE-TO-DEST-1
<== Posting #<PROCEDURE FLY-PLANE-TO-DEST-1> for :EXECUTE ==>
<== Trigger: ((AT (QUOTE C5-1) (QUOTE DELTA))) ==>
 0~01:54:42...Suspending task FLY-PLANE-TO-DEST-1
              of FLY-TRANSPORT-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******
```

At the end of cycle 45 we see that processing of the fly-plane-to-dest-1 procedure has continued with the selection of the liftoff-fly task. The procedure liftoff-fly-1 was selected and posted to the agenda for processing by the Execute capability, and once again the fly-plane-to-dest-1 procedure was suspended. In cycle 46, the Execute capability begins processing the liftoff-fly-1 procedure.

```
####################################################
                Agenda (cycle:   46)


<AE-47   :  95 (EXECUTE #<PROCEDURE LIFTOFF-FLY-1>)>

--------------- Untriggered Entries ---------------
<AE-48   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

####################################################

:KS-EXECUTE0 *********************************
<AE-47/0:
     PRIORITY: 95
     TRIGGER: T
     BODY: (EXECUTE #<PROCEDURE LIFTOFF-FLY-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving LIFTOFF-FLY-1(PRIMITIVE) to Execute...
 LIFTOFF-FLY-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status R.
 0~01:54:43...Considering task LIFTOFF-FLY-1 with status R of
              FLY-PLANE-TO-DEST-1
              LIFTOFF-FLY-1 is a PRIMITIVE task.
<== Posting #<PROCEDURE LIFTOFF-FLY-1> for :RESOURCE ==>
 0~01:54:43...Dispatching task LIFTOFF-FLY-1 of FLY-PLANE-TO-DEST-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******

...The Resource capability was processed in cycle 47...
```

```
...cycle 48...
:KS-DISPATCHO **********************************
 0~01:54:43...(FLY C5-1 CITY-K DELTA) dispatched.
:KS-DISPATCHO ****** END OF KS-DISPATCH ******
```

At 1 hour 54 minutes (cycle 48) we see that the REA has dispatched the liftoff-fly-1 procedure to the environment as (fly c5-1 city-k delta) which means that the c5-1 resource is now in flight between City-K and Delta.

At initialization the REA created two active behaviors. One to update the REA's World Model concerning road information and the other to update the World Model concerning known plane resources (road information is updated once every hour and plane information once every two hours).

We see that at time 2 hours, day 0 that both behaviors are activated. This occurs during the flight of the c5-1 resource to Delta. At time 3 hours we see that the road information is once again updated by the active behavior update-road-information.

```
 0~02:00:03...Performing active behavior
              UPDATE-PLANE-RESOURCE-INFORMATION.
 0~02:00:03...Performing active behavior UPDATE-ROAD-INFORMATION.
 0~03:00:04...Performing active behavior UPDATE-ROAD-INFORMATION.
```

After several cycles have passed, processing reports from executing tasks and the active behaviors, we rejoin the execution of the fly-plane-to-dest-1 procedure. At time 3 hours, 6 minutes the C5-1 reports that it is at Delta. This triggers the processing of the fly-plane-to-dest-1 procedure again in cycle 64.

```
...cycles 49 to 63 used by sensor reports from the active behaviors...

##################################################
                Agenda (cycle:   64)

<AE-48   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>

--------------- Untriggered Entries ---------------
<AE-36   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################
```

```
:KS-EXECUTE0 **********************************
<AE-48/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT-GROUP ((AT (QUOTE C5-1) (QUOTE DELTA))))
     BODY: (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-PLANE-TO-DEST-1(NETWORK) to Execute...
 FLY-PLANE-TO-DEST-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~03:06:20...Considering task FLY-PLANE-TO-DEST-1 with status P of
              FLY-TRANSPORT-1
 0~03:06:20...Network of FLY-PLANE-TO-DEST-1 is:
              (#<SCHEMA NODE-36.0 LAND>)
 0~03:06:20...Eligible procedures are: (#<PROCEDURE LAND-1>)
<== Posting #<PROCEDURE LAND-1> for :EXECUTE ==>
 0~03:06:20...Creating task LAND-1 of FLY-PLANE-TO-DEST-1
<== Posting #<PROCEDURE FLY-PLANE-TO-DEST-1> for :EXECUTE ==>
<== Trigger: ((PARKED-AT-GATE (QUOTE C5-1) (QUOTE YES))) ==>
 0~03:06:20...Suspending task FLY-PLANE-TO-DEST-1
              of FLY-TRANSPORT-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******
```

The Execute capability selects the remaining task of fly-plane-to-dest-1's network (i.e., land), chooses a procedure for the land task (i.e., land-1), and posts it to the agenda. Once again the execution of the fly-plane-to-dest-1 procedure is suspended. Execution continues in cycle 65.

```
####################################################
               Agenda (cycle:   65)


<AE-66   :  95 (EXECUTE #<PROCEDURE LAND-1>)>

--------------- Untriggered Entries --------------
<AE-67   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

####################################################

:KS-EXECUTE0 **********************************
<AE-66/0:
     PRIORITY: 95
     TRIGGER: T
```

```
     BODY: (EXECUTE #<PROCEDURE LAND-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving LAND-1(NETWORK) to Execute...
 LAND-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status R.
 0~03:06:20...Considering task LAND-1 with status R of
              FLY-PLANE-TO-DEST-1
              LAND-1 is a NETWORK task.
              The network is ((TOUCHDOWN C5-1 DELTA) (GOTO-GATE C5-1))
<== Posting #<PROCEDURE TOUCHDOWN-1> for :EXECUTE ==>
 0~03:06:20...Creating task TOUCHDOWN-1 of LAND-1
<== Posting #<PROCEDURE LAND-1> for :EXECUTE ==>
<== Trigger: ((RES-STATUS (QUOTE C5-1) (QUOTE ON-GROUND))) ==>
 0~03:06:21...Suspending task LAND-1
              of FLY-PLANE-TO-DEST-1
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

In cycle 65, the Execute capability begins processing of the land-1 procedure. It determines that it is a network procedure, therefore it must decompose it into the tasks of touchdown and goto-gate. It selects the procedure touchdown-1 as the only presently eligible task to execute, posts it to the agenda, and suspends the further execution of the land-1 task.

```
#####################################################
                Agenda (cycle:   66)


<AE-68   :  95 (EXECUTE #<PROCEDURE TOUCHDOWN-1>)>

--------------- Untriggered Entries ---------------
<AE-69   :  95 (EXECUTE #<PROCEDURE LAND-1>)>
<AE-67   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-36   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

#####################################################

:KS-EXECUTEO **********************************
<AE-68/0:
     PRIORITY: 95
     TRIGGER: T
     BODY: (EXECUTE #<PROCEDURE TOUCHDOWN-1>)
     CONTROLLER-INFO: (C5-1)>
```

```
:KS-EXECUTE Receiving TOUCHDOWN-1(PRIMITIVE) to Execute...
:KS-EXECUTE TOUCHDOWN-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
            and has status R.
 0~03:06:21...Considering task TOUCHDOWN-1 with status R of LAND-1
             TOUCHDOWN-1 is a PRIMITIVE task.
<== Posting #<PROCEDURE TOUCHDOWN-1> for :RESOURCE ==>
 0~03:06:21...Dispatching task TOUCHDOWN-1 of LAND-1
:KS-EXECUTEO ****** END OF KS-EXECUTE ******

...The Resource capability was processed in cycle 67...

...Touchdown-1 procedure dispatched in cycle 68...

...World Model updated from sensor reports in cycle 69...

...Land-1 procedure again processed by Execute capability.
   Goto-gate-1 procedure selected for goto-gate task in cycle
   70...

...The Resource capability was processed in cycle 71...

...Goto-gate-1 procedure dispatched in cycle 72...

...World Model updated from sensor reports in cycles 73 and 74...
```

Once the C5-1 resource has landed and taxied to the gate it reports that it is parked at the gate. This fact triggers the processing of the several tasks in cycle 75. The fly-transport-1 procedure which has been waiting on the agenda for the fly-plane-to-dest-1 procedure to complete is triggered by the fact that the C5-1 resource is now reported to be at its destination and parked at the gate. In addition, the fly-plane-to-dest-1 and land-1 procedures are also triggered by this same information. The Agenda Manager chooses the intention which has been processing the longest (i.e., the one with the lowest AE number).

```
##################################################
               Agenda (cycle:   75)

<AE-36  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-67  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-74  :  95 (EXECUTE #<PROCEDURE LAND-1>)>

-------------- Untriggered Entries --------------
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
##################################################
```

```
:KS-EXECUTEO **********************************
<AE-36/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT-GROUP
                ((AND (AT (QUOTE C5-1) (QUOTE DELTA))
                      (PARKED-AT-GATE (QUOTE C5-1) (QUOTE YES)))))
     BODY: (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-TRANSPORT-1(NETWORK) to Execute...
 FLY-TRANSPORT-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~03:19:08...Considering task FLY-TRANSPORT-1 with status P
              of (FLY-TRANSPORT C5-1 DELTA CITY-K)
 0~03:19:08...Network of FLY-TRANSPORT-1 is:
              (#<SCHEMA NODE-18.0 ACT-SENSOR>
               #<SCHEMA NODE-19.0 UNLOAD>
               #<SCHEMA NODE-20.0 REFUEL-PLANE>)
 0~03:19:08...Eligible procedures are: (#<PROCEDURE ACT-SENSOR-1>
                                        #<PROCEDURE UNLOAD-2>)
<== Posting #<PROCEDURE ACT-SENSOR-1> for :EXECUTE ==>
 0~03:19:08...Creating task ACT-SENSOR-1 of FLY-TRANSPORT-1
<== Posting #<PROCEDURE UNLOAD-2> for :EXECUTE ==>
 0~03:19:08...Creating task UNLOAD-2 of FLY-TRANSPORT-1
<== Posting #<PROCEDURE FLY-TRANSPORT-1> for :EXECUTE ==>
<== Trigger: ((AND (LOAD-STATUS (QUOTE C5-1) (QUOTE EMPTY))
                   (RES-STATUS (QUOTE C5-1) (QUOTE AVAILABLE)))
                   (AND (TIME-STAMP (QUOTE (SENSOR ACT-SENSOR C5-1))
                          (QUOTE ?TIME))
                        (<= (* (- (GET-UNIVERSAL-TIME) (QUOTE ?TIME))
                          DIARY::*SIMULATED-SECONDS-PER-SECOND*)
                          *ACT-SENSOR-FREQ*))) ==>
 0~03:19:09...Suspending task FLY-TRANSPORT-1
              of (FLY-TRANSPORT C5-1 DELTA CITY-K)
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

The execution of the fly-transport-1 procedure, having been triggered by the fact that the effects of the fly-plane-to-dest-1 procedure have been achieved, continues with processing by the Execute capability. It selects the next task eligible (i.e., act-sensor) to execute from fly-transport-1's network, posts its procedure to the agenda, and again suspends the execution of fly-transport-1. Execution of the fly-plane-to-dest-1 procedure continues in cycle 76.

```
##################################################
              Agenda (cycle:   76)

<AE-67  :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-74  :  95 (EXECUTE #<PROCEDURE LAND-1>)>
<AE-78  :  95 (EXECUTE #<PROCEDURE ACT-SENSOR-1>)>
<AE-79  :  95 (EXECUTE #<PROCEDURE UNLOAD-2>)>

--------------- Untriggered Entries --------------
<AE-80  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

:KS-EXECUTEO **********************************
<AE-67/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT
                ((PARKED-AT-GATE (QUOTE C5-1) (QUOTE YES))))
     BODY: (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving FLY-PLANE-TO-DEST-1(NETWORK) to Execute...
 FLY-PLANE-TO-DEST-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
    and has status P.
 0~03:19:09...Considering task FLY-PLANE-TO-DEST-1 with status P of
               FLY-TRANSPORT-1
 0~03:19:09...Network of FLY-PLANE-TO-DEST-1 is empty.
 0~03:19:09...FLY-PLANE-TO-DEST-1 has completed.
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

In cycle 76, the fly-plane-to-dest-1 procedure is processed by the Execute capability. It is determined that its network has been completely executed and that its effects have been achieved. No further processing takes place for fly-plane-to-dest-1. Execution continues in cycle 77.

```
##################################################
              Agenda (cycle:   77)

<AE-74  :  95 (EXECUTE #<PROCEDURE LAND-1>)>
<AE-78  :  95 (EXECUTE #<PROCEDURE ACT-SENSOR-1>)>
<AE-79  :  95 (EXECUTE #<PROCEDURE UNLOAD-2>)>

--------------- Untriggered Entries --------------
<AE-80  :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
```

```
##################################################

:KS-EXECUTEO **********************************
<AE-74/0:
     PRIORITY: 95
     TRIGGER: (WAIT-ON-EFFECT
                ((PARKED-AT-GATE (QUOTE C5-1) (QUOTE YES))))
     BODY: (EXECUTE #<PROCEDURE LAND-1>)
     CONTROLLER-INFO: (C5-1)>

 Receiving LAND-1(NETWORK) to Execute...
 LAND-1 is servicing #<SCHEMA NODE-1.0 FLY-TRANSPORT>
                and has status P.
 0~03:19:09...Considering task LAND-1 with status P of
              FLY-PLANE-TO-DEST-1
 0~03:19:10...Network of LAND-1 is empty.
 0~03:19:10...LAND-1 has completed.
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

In cycle 77, we come to the end of this detailed example. The Execute capability determines that the land-1 task has successfully completed and that it no longer requires processing. Execution does continue in cycle 78 with the processing of the unload-2 task; however, we will end our introspection here.

There is an enormous amount of detail in the simple task we have just examined. This is due to the intermediate processing of subtasks and reports from the environment, as well as, trying to describe a complete picture by examining the agenda management and reasoning processes. In the remaining examples we shall not provide so much detail in order to reduce the space requirements unless such detail is warranted for clarification.

8.2 The Three-star Rating

In Chapter 7, we saw the criteria for assigning a '***' rating to each of the characterized abilities. In this section we will examine the execution behavior of the REA design to determine if each of these criteria is met, and if not then discuss why this is the case.

In order to achieve this, we examine these criteria in the context of the Small Scale NEO discussed in Section 6.3.1. We describe each characteristic in a separate section,

beginning with guaranteed response.

8.2.1 Guaranteed Response

In Section 6.5.1 it was stated that in order to achieve guaranteed response, the REA would have to (1) act with single task directive, (2) act without any task directive, and (3) act with multiple task directives. Each of these situations is discussed in the following sections.

Acting with a Single Task Directive

Acting with a single task directive requires that the REA receives an IACL synthesize message, understand the contents of the message, synthesize a Task Directive Object (TDO), and begin to execute the tasks of the task directive according to specified ordering constraints. If we look at the following snapshot of execution for the Small Scale NEO Scenario we can see that the REA does indeed possess the ability to act with a single task directive according to these requirements.

```
:KS-SYNTHESIZE0 **********************************
 0~00:00:12...Synthesizing a new Task Directive...
 0~00:00:12...T-DIRECTIVE-1 has been synthesized.  It looks like:
Name:     T-DIRECTIVE-1
Status:   U
Priority: 15
Processing: NIL
Processed: NIL
Task:
 Name: NODE-3(1.0)          I-type: FLY-TRANSPORT      E-Status: U
 Name: NODE-4-1(2.0)        I-type: DRIVE              E-Status: U
 Name: NODE-4-2(3.0)        I-type: LOAD               E-Status: U
 Name: NODE-4-3(4.0)        I-type: DRIVE              E-Status: U
 Name: NODE-4-4(5.0)        I-type: UNLOAD             E-Status: U
 Name: NODE-5-1(6.0)        I-type: DRIVE              E-Status: U
 Name: NODE-5-2(7.0)        I-type: LOAD               E-Status: U
 Name: NODE-5-3(8.0)        I-type: DRIVE              E-Status: U
 Name: NODE-5-4(9.0)        I-type: UNLOAD             E-Status: U
 Name: NODE-6-1(10.0)       I-type: DRIVE              E-Status: U
 Name: NODE-6-2(11.0)       I-type: LOAD               E-Status: U
 Name: NODE-6-3(12.0)       I-type: DRIVE              E-Status: U
 Name: NODE-6-4(13.0)       I-type: UNLOAD             E-Status: U
 Name: NODE-7(14.0)         I-type: FLY-PASSENGERS     E-Status: U
```

```
 Name: NODE-8(15.0)          I-type: FLY-TRANSPORT       E-Status: U
:KS-SYNTHESIZEO ****** END OF KS-SYNTHESIZE ******

:KS-SUPERVISEO **********************************
 KS-SUPERVISE: Receiving T-DIRECTIVE-1(R) to Supervise...
 Schemas eligible to execute are : (#<SCHEMA NODE-1.0 FLY-TRANSPORT>)
 Schemas cleared to execute are : (#<SCHEMA NODE-1.0 FLY-TRANSPORT>)
 Selecting #<PROCEDURE FLY-TRANSPORT-1> of
           #<SCHEMA NODE-1.0 FLY-TRANSPORT>
:KS-SUPERVISEO ****** END OF KS-SUPERVISE ******
```

Here we see that the IACL synthesize message is received, the TDO is synthesized, and the TDO is then passed on to the Supervise capability which selects the procedure fly-transport-1 for execution.

Acting without a Task Directive

Acting without a Task Directive is possible by either the triggering of a passive behavior or by the presence of an active behavior. Here we will consider how this can be achieved through the use of an active behavior, but later we will see how this could just as well have been achieved with a passive behavior.

For the NEO scenarios, the REA is given two active behaviors at initialization that help keep its World Model updated. These are update-road-information and update-plane-resource-information. The "road" behavior becomes active every hour. Upon activation it randomly selects a known road on the island of Pacifica and requests information about that road. The "plane" behavior is activated every two hours and requests information about a particular plane resource that is randomly selected. This behavior uses the sensor models to determine the appropriate sensor to request information from based upon the type of plane resource it has selected (i.e., either air-cargo-transport or air-passenger-transport). Below we see the creation of these behaviors at initialization and notification of their activations at various time points during execution.

```
The Active Behavior: UPDATE-ROAD-INFORMATION was created.
The Active Behavior: UPDATE-PLANE-RESOURCE-INFORMATION was created.
                              .
                              .
                              .
```

```
0~01:00:03...Performing active behavior UPDATE-ROAD-INFORMATION.
0~02:00:03...Performing active behavior
             UPDATE-PLANE-RESOURCE-INFORMATION.
0~02:00:03...Performing active behavior UPDATE-ROAD-INFORMATION.
0~03:00:04...Performing active behavior UPDATE-ROAD-INFORMATION.
0~04:00:04...Performing active behavior
             UPDATE-PLANE-RESOURCE-INFORMATION.
0~04:00:04...Performing active behavior UPDATE-ROAD-INFORMATION.
```

Acting with Multiple Task Directives

Acting with multiple Task Directives is similar to that of acting with a single one from the standpoint of the REA. The difference is that it has the tasks of multiple Task Directives on the agenda at one time. We can see this by looking at an execution snapshot when several TDOs are synthesized or by looking at the agenda during execution. The execution snapshot simply tells us the time when a TDO was synthesized and gives us the TDO information. For instance:

```
0~00:00:08...Synthesizing a new Task Directive...
0~00:00:08...T-DIRECTIVE-1 has been synthesized.  It looks like:
                              .
                              .
                              .
0~00:00:20...Synthesizing a new Task Directive...
0~00:00:20...T-DIRECTIVE-60 has been synthesized.  It looks like:
```

The agenda snapshot on the other hand shows us the fact that the Task Directives are being processed concurrently. For instance:

```
#####################################################
                Agenda (cycle:   33)

<AE-35   :  85 (EXECUTE #<PROCEDURE ACT-SENSOR-1>)>

--------------- Untriggered Entries ---------------
<AE-36   :  85 (EXECUTE #<PROCEDURE LOAD-CARGO-TRANSPORT-3>)>
<AE-33   :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-60>)>
<AE-26   :  95 (EXECUTE #<PROCEDURE LOAD-2>)>
<AE-8    :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-6    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

#####################################################
```

Either way, we can see that the REA possesses the ability to act with multiple Task Directives.

8.2.2 Failure Recovery

The criteria for the failure recovery characteristic were described in Section 6.5.1. Here we show, by example, that the REA design meets these criteria.

Recovery from an Exogenous Event

In order to illustrate the ability of the REA to recover from some exogenous events we must consider an execution trace and a snapshot of the simulation history from the Pacifica Simulator. We begin our discussion just after the ground transport resources have been successfully unloaded from the c5-1 resource at time 4 hours 54 minutes. (Figure 8.1).

```
4:54:24 Completed: (Unload-Plane c5-1 at delta)
4:54:26 (S) Sensing: c5-1 with ACT-Sensor...
4:54:30 Started: (Drive gt1 from delta
                  to calypso on road-cd at 54)
4:54:30 Started: (Drive gt2 from delta
                  to barnacle on road-bd at 46)
4:56:07 FAILURE: (Drive Gt1 from Delta
                  to Calypso on Road-Cd at 54)
4:56:07 (I) Gt1 traveled 1 miles of 50 along Road-Cd
            before the FAILURE: Broken-Fan-Belt!
4:56:09 Started: (GT-Mech-Tow-and-Repair of gt1 to delta)
5:00:05 (S) Sensing: road-bd with Road-Sensor...
5:04:42 (E) Climate in Abyss is now Stormy.
5:09:25 (E) Climate in Delta is now Rainy.
5:16:40 (E) Gt2 ATTACKED by T-Fal-1
             Mechanical status of is now Poor.
5:28:09 Completed: (GT-Mech-Tow-and-Repair of gt1 to delta)
5:28:09 (I) Mechanical Status of gt1 is now Good.
5:28:12 Started: (Drive gt1 from delta
                  to calypso on road-cd at 54)
5:28:12 (I) Road conditions warrant a speed of 37 m.p.h.
```

Figure 8.1: Simulation History Snapshot

At approximately 4 hours, 56 minutes into the simulation we intervene and cause the fan belt on the gt1 ground transport to break. This failure is immediately reported to the REA. The Failure capability determines that no specialist failure handlers for a broken fan belt exist so, it looks for some general knowledge which might be used to address the failure.

In this case, it finds some general knowledge for addressing this particular failure in the form of the fix-gt-mechanical-on-road task. It selects the fix-gt-mr-1 procedure, and at time 4 hours, 56 minutes dispatches the task to begin the repair of the fan belt.

```
 0~04:54:29...(DRIVE GT1 DELTA CALYPSO ROAD-CD 54) dispatched.

:KS-FAILURE0 **********************************
 Receiving failure event...
 The reason for failure was: BROKEN-FAN-BELT
 Checking for specialist to handle BROKEN-FAN-BELT
 No specialist found.  Checking general knowledge...
:KS-FAILURE0 ******** END OF KS-FAILURE *******

 0~04:56:09...Considering task FIX-GT-MR-1 with status R
              of (FIX-GT-MECHANICAL GT1 GEO-LOCATION)
              FIX-GT-MR-1 is a PRIMITIVE task.
 0~04:56:09...Dispatching task FIX-GT-MR-1 to address FAILURE.
 0~04:56:09...(GT-MECH-TOW GT1) dispatched.
```

The fix-gt-mr-1 task, dispatched as gt-mech-tow, causes the resource involved in the failure to be towed to the nearest city where it can be repaired. If the resource has traveled less than or equal to fifty percent of the total distance of the road then the resource is towed back to the last city it was located. Otherwise, the resource is towed to the next city on the road it was traveling. At the point of the failure in this example, the gt1 resource had only traveled one of 50 miles along Road-CD (see Figure 8.1). Therefore, it was towed back to Delta and repairs were made there.

```
 0~05:28:11...Considering task DRIVE-1 with status P
              of (DRIVE GT1 CALYPSO DELTA)
 0~05:28:11...Network of DRIVE-1 is:
              (#<SCHEMA NODE-54.0 GO-LOCATION>
               #<SCHEMA NODE-55.0 CITY-SENSOR>)
              Checking for repeat on DRIVE-1 since all previously
              dispatched subtasks did not execute successfully...
```

```
0~05:28:11...Repeating task (DRIVE GT1 CALYPSO DELTA)
             Since its effects were not achieved.
0~05:28:11...Considering task DRIVE-1 with status R
             of (DRIVE GT1 CALYPSO DELTA)
             DRIVE-1 is a NETWORK task.
             The network is
              ((GO-LOCATION GT1 DELTA CALYPSO ROAD-CD 54)
               (CITY-SENSOR CALYPSO))
0~05:02:11...Creating task GO-LOCATION-1 of DRIVE-1
0~05:28:11...Suspending task DRIVE-1
             of (DRIVE GT1 CALYPSO DELTA)
0~05:28:11...Considering task GO-LOCATION-1 with status R of DRIVE-1
             GO-LOCATION-1 is a PRIMITIVE task.
0~05:28:11...Dispatching task GO-LOCATION-1 of DRIVE-1
0~05:28:12...(DRIVE GT1 DELTA CALYPSO ROAD-CD 54) dispatched.
```

Once the repairs have been completed at time 5 hours, 28 minutes the REA attempts to repeat the task that failed. If the task is repeatable, and the conditions are such that it can be repeated (as they are in this example) then normal execution of the Task Directive continues from the failure as if nothing had happened. If on the other hand, the task were not repeatable or the conditions were not right to repeat the task then the planning agent would have been notified.

This example has shown that the design allows the REA to respond to a failure, repair that failure, and continue normal execution after the repairs have successfully completed.

Detection of a Potential Failure by Protection Monitor

In this example, we consider the benefits of causal structure based protection monitors and active sensing. We join the execution 14 minutes into the execution trace as a Task Directive is being synthesized after the REA has received an IACL synthesize message. Here we see that five active behaviors are created during the processing by the Synthesize capability.

```
0~00:14:10...Synthesizing a new Task Directive...
The Active Behavior: UPDATE-5 was created.
The Active Behavior: UPDATE-8 was created.
The Active Behavior: UPDATE-11 was created.
The Active Behavior: UPDATE-18 was created.
```

```
The Active Behavior: UPDATE-24 was created.
0~00:14:20...T-DIRECTIVE-1 has been synthesized.
```

These correspond to causal structure records from the IACL Synthesize message for the Small Scale NEO plan that the REA is to execute (see Appendix A). That is, an active behavior is created to periodically issue sensor requests to update the REA's World Model for each unique datum of a particular resource. For this particular example, the location and status of each ground transport, and the location of the c5-1 air cargo transport are to be kept updated. Hence, the five active behavior objects are created.

The frequency of each active behavior object is determined by the sensor that must be used to update the required information. Here, only two sensors are necessary to gather the information— gt-sensor and act-sensor. For the model of these sensors, the REA determines that the updates to the ground transport information should occur every 45 minutes, and the air cargo information should be updated every 30 minutes. Once created, each active behavior object is passed to the Active Behavior Manager (Section 4.3.6).

Also, during the synthesis process, protection monitors are established for each causal structure record (Section 5.3). Each time the World Model is updated, those monitors monitoring the updated information are examined to determine whether a violation has occurred. This is a violation in the sense that the new value is not what the monitor had expected it to be at that particular point in the execution.

```
0~08:20:07...###  Completed task (UNLOAD GT1 DELTA) of T-DIRECTIVE-1
0~08:20:09...Considering Task Directive T-DIRECTIVE-1 with status P
0~08:20:53...Considering task UNLOAD-1 with status P
             of (UNLOAD GT1 DELTA)
0~08:20:57...Network of UNLOAD-1 is empty.
0~08:20:58...UNLOAD-1 has completed.
0~08:21:02...Considering Task Directive T-DIRECTIVE-1 with status R
0~09:09:37...Performing active behavior
              UPDATE-PLANE-RESOURCE-INFORMATION.
0~09:09:43...Performing active behavior UPDATE-ROAD-INFORMATION.
0~09:29:53...Considering task DRIVE-1 with status P
             of (DRIVE GT2 DELTA BARNACLE)
0~09:29:56...Network of DRIVE-1 is:
             (#<SCHEMA NODE-105.0 GO-LOCATION>
              #<SCHEMA NODE-106.0 CITY-SENSOR>)
```

```
            Checking for repeat on DRIVE-1 since all previously
            dispatched subtasks did not execute successfully...
0~09:30:00...Repeating task (DRIVE GT2 DELTA BARNACLE)
            Since its effects were not achieved.
0~09:30:09...Considering task DRIVE-1 with status R
            of (DRIVE GT2 DELTA BARNACLE)
            DRIVE-1 is a NETWORK task.
            The network is
             ((GO-LOCATION GT2 BARNACLE DELTA ROAD-CD 54)
              (CITY-SENSOR DELTA))
0~09:30:14...Creating task GO-LOCATION-1 of DRIVE-1
0~09:30:15...Suspending task DRIVE-1
            of (DRIVE GT2 DELTA BARNACLE)
0~09:30:21...Considering task GO-LOCATION-1 with status R of DRIVE-1
            GO-LOCATION-1 is a PRIMITIVE task.
0~09:30:24...Dispatching task GO-LOCATION-1 of DRIVE-1
0~09:30:32...(DRIVE GT2 BARNACLE DELTA ROAD-CD 54) dispatched.
0~09:53:45...Protection VIOLATION detected by: MONITOR-4
            Value is CALYPSO not the expected value of DELTA
0~09:53:48...Protection VIOLATION detected by: MONITOR-6
            Value is CALYPSO not the expected value of DELTA
```

Now we jump approximately 8 hours into the execution trace to the point where the gt1 ground transport has completed the unloading of the nationals from Calypso at Delta. According to the plan, the gt1 resource is now to remain at Delta until the remaining nationals arrive in Delta and are loaded onto the B707. However, to demonstrate the ability of the REA to detect potential failures, at time 8 hours, 39 minutes we intervene in the simulation and move the gt1 ground transport to the city of Calypso. This fact is not reported to the REA since a failure has not occurred. Therefore, execution continues normally with the gt2 ground transport driving back to Delta along roads BC and CD.

With active sensing off the only hope of detecting the fact that the gt1 resource is not in Delta as it should be is when the REA dispatches the load-cargo-transport task and it fails because gt1 is not present. That failure would occur at 22 hours, 31 minutes into the execution. With active sensing on we can guarantee that the movement of the gt1 resource will be detected, at worst, in 90 minutes. It is actually detected at 9 hours, 53 minutes into execution by monitors 4 and 6, 1 hour and 14 minutes after it was moved. These violations result in IACL execution-failure messages being sent to the planning agent to notify it of the situation.

```
 8:14:00 Completed: (Load-Vehicle gt2 at barnacle with passengers)
 8:15:50 Started: (Drive gt2 from barnacle
                   to delta on road-bc at 46)
 8:15:50 (I) Road conditions warrant a speed of 37 m.p.h.
 8:19:00 Completed: (Unload-Vehicle gt1 at delta)
 8:20:00 (S) Sensing: gt1 with GT-Sensor...
 8:21:05 (E) Climate in City-K is now Sunny.
 8:22:12 (E) Gt1 now has 35.1 gallons of fuel.
 8:39:47 (S) Sensing: c5-1 with ACT-Sensor...
 9:06:19 (S) Sensing: gt2 with GT-Sensor...
 9:07:23 (E) Bay-Bridge ATTACKED by T-Fal-1
              Status is now Closed.
 9:09:45 (S) Sensing: c5-2 with ACT-Sensor...
 9:09:51 (S) Sensing: road-ad with Road-Sensor...
 9:11:08 (E) Climate in Exodus in now Sunny.
 9:29:06 Completed: (Drive gt2 from barnacle
                     to delta on road-bc at 37)
 9:30:34 Started: (Drive gt2 from barnacle
                   to delta on road-cd at 54)
 9:40:41 (S) Sensing: c5-1 with ACT-Sensor...
 9:52:47 (S) Sensing: gt1 with GT-Sensor...
10:11:08 (E) Climate in Delta is now Rainy.
10:32:56 Completed: (Drive gt2 from barnacle
                     to delta on road-cd at 54)
10:34:51 Started: (Unload-Vehicle gt2 at delta)
10:38:33 (S) Sensing: gt2 with GT-Sensor...
10:42:26 (S) Sensing: c5-1 with ACT-Sensor...
10:56:48 Completed: (Unload-Vehicle gt2 at delta)
10:58:33 Started: (Refuel gt2)
                  .
                  .
22:25:48 Completed: (Unload-Vehicle gt2 at delta)
22:26:48 (S) Sensing: gt2 with GT-Sensor...
22:26:55 (S) Sensing: c5-1 with ACT-Sensor...
22:29:50 Started: (Pre-Flight of b707)
22:31:03 (E) Mechanical status of C5-1 is now bad.
22:31:33 Started: (Load-Cargo c5-1 at delta with gts-only)
22:31:45 (S) Sensing: road-ae with Road-Sensor...
22:34:48 Completed: (Pre-Flight of b707)
22:36:24 Started: (Taxi b707)
```

Figure 8.2: Simulation History Snapshot

The protection monitors with active sensing on effectively gives the planning agent 12 hours to develop a repair plan.

Recovery from a Failure via Behavior Mechanism

To demonstrate this capability in the design, we add a passive behavior to the REA whose purpose is to monitor for problems related to plane resources (i.e., both air-cargo and air-passenger transports). Specifically, this behavior is triggered when it detects that the mechanical status for a plane is "bad," or when a plane's tire status is "blown." Upon activation this behavior causes a capability called Plane-Failure to process the event and repair the affected resource.

```
####################################################
               Agenda (cycle:    1)

 No Triggered entries...

--------------- Untriggered Entries ---------------
<AE-2    : 195 (PLANE-FAILURE (EFFECTS NIL PLANE-TIRE-OR-MECH-PROBLEM)
               #<Passive-Behavior #X13B61DE>)>

####################################################

 0~00:36:39...Synthesizing a new Task Directive...
 0~00:46:43...(LOAD-CARGO C5-1 CITY-K GTS-ONLY) dispatched.
```

We begin examining the execution in cycle 1. Here we see that the innate passive behavior we have added for this example is installed as an untriggered agenda entry (i.e., intention). At 36 minutes into the simulation we send an IACL synthesize message to the REA which contains the NEO plan. The REA then begins to execute that plan.

Approximately 30 minutes into the simulation we manually intervene and tell the Pacifica Simulator to burst a tire on the B707 air passenger transport that is located in Delta, Pacifica (Figure 8.3). This particular action, since it is not a task failure, is not reported to the REA. Therefore, the only way for the REA to realize that the tire of the B707 is blown is either by requesting a status report from the B707 (i.e., sensing) or by attempting to use the B707 for some task, at which time the blown tire

```
   40:46 (S) Sensing: gt1 with GT-Sensor...
   41:44 (S) Sensing: gt2 with GT-Sensor...
   44:28 (S) Sensing: c5-1 with ACT-Sensor...
   46:46 Started: (Load-Cargo c5-1 at city-k with gts-only)
 1:04:11 (S) Sensing: road-cd with Road-Sensor...
 2:03:21 (S) Sensing: b707 with APT-Sensor...
 2:04:40 (E) Bay-Bridge ATTACKED by T-Fal-1
             Status is now Closed.
 2:05:26 Started: (Plane-Tire-Repair on b707 at delta)
 2:05:43 (S) Sensing: road-ad with Road-Sensor...
 2:17:07 (E) Volcanic activity...Road-BD is now closed.
 2:35:06 Completed: (Load-Cargo c5-1 at city-k with gts-only)
 2:38:17 Started: (Taxi c5-1)
 2:43:41 Completed: (Taxi c5-1)
 2:45:13 Started: (Tower-Clearance c5-1)
 2:47:24 Completed: (Tower-Clearance c5-1)
 2:48:28 (E) Volcanic activity...Road-BD is now closed.
 2:48:53 Started: (Fly c5-1 from city-k to delta)
 2:48:53 (E) Bay-Bridge ATTACKED by T-Fal-1
             Status is now Closed.
 2:50:26 (E) Climate in Calypso is now Sunny.
 3:06:23 Completed: (Plane-Tire-Repair on b707 at delta)
 3:06:23 (I) Tire Status of b707 is now Okay.
 3:06:39 (S) Sensing: road-ab with Road-Sensor...
 3:07:46 (E) Climate in Abyss is now Sunny.
```

Figure 8.3: Simulation History Snapshot

would be reported as a failure.

In the Small Scale NEO Scenario, the B707 is initially located in Delta, Pacifica and is not expected to be used until the ground transports have transported all of the nationals from their various locations around the island to Delta. This period of time is minimally 14 hours, and could be much greater depending upon the number (and type) of exogenous events. Another problem is that no causal structure information is given in the IACL synthesize message related to the B707 resource, so active sensing would not help to detect this situation. Instead, we rely upon the innate active behavior of the REA that periodically updates its World Model regarding plane resource information.

```
0~01:04:05...Performing active behavior UPDATE-ROAD-INFORMATION.
0~02:03:15...Performing active behavior
             UPDATE-PLANE-RESOURCE-INFORMATION.
```

At time 2 hours, 3 minutes the REA selects a plane resource from which to gather information with the activation of the update-plane-resource-information behavior. Here the selected resource is the B707, so a sensor request is dispatched for the apt-sensor. When the information from the apt-sensor has been assimilated into the World Model this causes the triggering of the passive behavior plane-tire-or-mech-problem. This behavior subsequently causes the specialist failure handler, Plane-Failure, to be placed on the agenda.

```
####################################################
                Agenda (cycle:   32)


<AE-28   : 195 (PLANE-FAILURE (EFFECTS NIL PLANE-TIRE-OR-MECH-PROBLEM)
               #<Passive-Behavior #X14D73BE>)>

--------------- Untriggered Entries ---------------
<AE-27   :  95 (EXECUTE #<PROCEDURE LOAD-2>)>
<AE-9    :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-7    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

####################################################

:KS-PLANE-FAILUREO **********************************
 Receiving failure event: PLANE-TIRE-OR-MECH-PROBLEM
:KS-PLANE-FAILUREO **** END OF KS-PLANE-FAILURE *****

The Passive Behavior: RE-ESTABLISH-BEHAVIOR was created.
```

In cycle 32, the Plane-Failure capability determines that the B707 has a blown tire. It then finds that appropriate knowledge to address the failure (i,e., fix-plane-tire) and dispatches a contextually valid procedure to begin the repair. The Plane-Failure capability also creates a passive behavior to re-establish the plane-tire-or-mech-problem passive behavior once the B707 has been successfully repaired.

```
####################################################
                Agenda (cycle:   33)


<AE-36   : 100 (EXECUTE #<PROCEDURE FIX-PLANE-T-1>)>

--------------- Untriggered Entries ---------------
<AE-37   :  10 (ESTAB-BEHAVIOR #<Passive-Behavior #X14D73BE>
```

```
                #<Passive-Behavior #X13D34CE>)>
<AE-27   :  95 (EXECUTE #<PROCEDURE LOAD-2>)>
<AE-9    :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-7    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################

 0~02:04:57...Considering task FIX-PLANE-T-1 with status R
              of (FIX-PLANE-TIRE B707 DELTA)
              FIX-PLANE-T-1 is a PRIMITIVE task.
 0~02:05:00...Dispatching task FIX-PLANE-T-1 to address FAILURE.
 0~02:05:23...(PLANE-TIRE-REPAIR B707 DELTA) dispatched.
 0~02:05:38...Performing active behavior UPDATE-ROAD-INFORMATION.
 0~02:36:47...Considering task FLY-TRANSPORT-1 with status P
              of (FLY-TRANSPORT C5-1 DELTA CITY-K)
 0~02:36:56...Eligible procedures are:
               (#<PROCEDURE FLY-PLANE-TO-DEST-1>)
 0~02:36:58...Creating task FLY-PLANE-TO-DEST-1 of FLY-TRANSPORT-1
 0~02:37:00...Suspending task FLY-TRANSPORT-1
              of (FLY-TRANSPORT C5-1 DELTA CITY-K)
 0~02:38:14...(TAXI C5-1) dispatched.
 0~02:45:09...(TOWER-CLEARANCE C5-1) dispatched.
 0~02:48:51...(FLY C5-1 CITY-K DELTA) dispatched.
 0~03:06:34...Performing active behavior UPDATE-ROAD-INFORMATION.
```

At 2 hours, 5 minutes the fix-plane-t-1 task is dispatched. During this time the load-cargo task dispatched at 46 minutes completes, and the execution of the fly-transport-1 task continues to execute.

According to the simulation history snapshot, the tire of the B707 is repaired at time 3 hours, 6 minutes. This fact causes the re-establish-behavior capability to become triggered in cycle 63 which, in turn, causes the plane-tire-or-mech-problem passive behavior to be re-established on the untriggered agenda seen in cycle 64.

```
##################################################
                Agenda (cycle:   63)

<AE-37   :  10 (ESTAB-BEHAVIOR #<Passive-Behavior #X14D73BE>
                #<Passive-Behavior #X13D34CE>)>

--------------- Untriggered Entries ---------------
<AE-59   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-47   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-7    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>
```

```
##################################################

##################################################
                Agenda (cycle:   64)

<AE-68   : 200 (WORLD (EFFECTS ((SENSOR ROAD-SENSOR ROAD-AB)) NIL))>

--------------- Untriggered Entries ---------------
<AE-69   : 195 (PLANE-FAILURE (EFFECTS NIL PLANE-TIRE-OR-MECH-PROBLEM)
                #<Passive-Behavior #X16574D6>)>
<AE-59   :  95 (EXECUTE #<PROCEDURE FLY-PLANE-TO-DEST-1>)>
<AE-47   :  95 (EXECUTE #<PROCEDURE FLY-TRANSPORT-1>)>
<AE-7    :  90 (SUPERVISE #<TASK-DIRECTIVE T-DIRECTIVE-1>)>

##################################################
```

Here we have seen how the behavior mechanism is able to detect and instigate a recovery from a failure. This example also illustrates how the specialist knowledge and passive behavior concepts are integrated to adapt and dynamically enhance the failure management capabilities of the REA.

Detection of a Failure Beyond the REA's Capabilities

Having discussed a similar situation previously, we will not discuss this capability in the same detail as the other examples. This capability presents itself when a failure is detected for which the REA does not possess specialist or general knowledge to address that failure.

For example, let's return for a moment to the first subsection of Section 8.2.2 where we were discussing how the REA is able to recover from exogenous events. There the failure was a broken fan belt on the gt1 ground transport. If the failure had been something like a broken clutch, the REA would again have tried to find some specialist or general knowledge concerning a broken clutch. However, unlike the previous example, since the REA does not possess any such knowledge it would have to fail. The failure to recovery from the broken clutch problem would cause the REA to request assistance from the planning agent since the ability to recover from the failure is beyond the REA's knowledge and/or capabilities.

This request for assistance would be in the form of an IACL execution-failure message. Once this message had been sent, the REA then would resume work on its other intentions. By sending the IACL message the REA transfers responsibility for the further execution of the failed plan to the planning agent.

This solution to managing failures which the REA does not know how to address may not be the best solution since it is essentially "passing the buck" to the planning agent, but it does allow the REA to gracefully move on from the problem. The tradeoff is whether to add the capability to allow the REA to locally search and develop a recovery plan itself based upon its knowledge of the environment or to prevent it from wasting time on problems it cannot solve. The approach taken in this design is the latter. However, the general framework of the REA does allow improved localized knowledge to be imported.

Detection of a Precondition Failure

Here we show how the design allows the REA to address the final criteria with regards to the failure recovery capability. We examine the situation were the REA possesses the knowledge to address a particular failure, but a precondition of applying that knowledge is not satisfied. This is called a precondition failure.

In this particular example, the REA possesses the knowledge to repair a hole in the fuel tank (Figure 8.4) of a ground transport when the transport is located in a city and is loaded with passengers. However, since the Domain Data Object definition of the repair stipulates the condition that the ground transport resource must be in a city and not traveling a road (as in the following example) the knowledge cannot be applied. In this situation the REA reports that it has no knowledge to address the detected failure to the planning agent via an IACL execute-failure message (Section 5.2.5).

We join the execution at the point in the execution of the Small Scale NEO plan where the ground transports—gt1 and gt2—have just been unloaded from the c5-1 cargo transport at Delta, Pacifica (i.e., the fly-transport task has successfully completed). The REA directs gt1 to travel to Calypso via Road-CD at 54 miles per hour, and gt2 to travel to Barnacle via Road-BD at 46 miles per hour. The road conditions of

```
(make-instance 'Domain-Data
               :name 'replace-fuel-tank
               :variables '(?res ?loc)
               :expands '(replace-fuel-tank ?res ?loc)
               :conditions '(o-type '?loc 'city)
               :procedures
               (list
                '(RFT-1 network
                  (context (and (at '?res '?loc)
                                (load-status '?res 'loaded)
                                (res-status '?res 'unavailable)
                                (fuel-level '?res 0)))
                  (network ((tn1 (unload ?res ?loc) ())
                            (tn2 (repair-fuel-tank ?res ?loc) ())
                            (tn3 (refuel ?res) ())
                            (tn4 (load ?res ?loc) ())))
                  (orderings ((tn1 nil (tn2))
                              (tn2 (tn1) (tn4))
                              (tn3 (tn1 tn2) nil)
                              (tn4 (tn1 tn2) nil)))))
               :effects '((res-status '?res 'available)
                          (fuel-level '?res (max-fuel '?res)))
               :uses-resources (list '?res)
               :exec-cost 4
               :end-cond '(and (res-status '?res 'available)
                               (fuel-level '?res (max-fuel '?res)))
               :duration '(80 100))
```

Figure 8.4: Replace-fuel-tank DDO

Road-BD dictate that gt1 will actually only be traveling at 37 miles per hour; however this information is not communicated to the REA.

At time 6 hours, 18 minutes in the simulation snapshot (Figure 8.5) an exogenous event occurs resulting in a hole in the fuel tank of the gt2 ground transport. This problem is reported to the REA and his processed by the Failure capability. The Failure capability determines that the REA does not possess specialist knowledge to address the failure, nor any general knowledge. In actuality, the REA does possess the knowledge to address the failure, but that knowledge is incomplete since it does not address the current circumstances surrounding the failure. In this situation, since the gt2 resource is traveling down Road-BD, the conditions for the application of the

```
5:31:08 (S) Sensing: b707 with APT-Sensor...
5:33:17 Completed: (Unload-Plane c5-1 at delta)
5:33:17 (E) Volcanic activity...Road-BD is now closed.
5:35:12 (S) Sensing: c5-1 with ACT-Sensor...
5:37:30 Started: (Drive gt1 from delta
                  to calypso on road-cd at 54)
5:39:50 (S) Sensing: barnacle with City-Sensor...
5:41:25 Started: (Drive gt2 from delta
                  to barnacle on road-cd at 46)
5:41:25 (I) Road conditions warrant a speed of 54 m.p.h.
5:43:58 (S) Sensing: gt1 with GT-Sensor...
6:06:14 (E) Bay-Bridge ATTACKED by T-Fal-1
             Status is now Closed.
6:06:24 (S) Sensing: c5-1 with ACT-Sensor...
6:08:31 (S) Sensing: road-ae with Road-Sensor...
6:18:38 FAILURE: (Drive Gt2 from Delta
                  to Barnacle on Road-Cd at 54)
6:18:38 (I) Gt2 traveled 33 miles of 50 along Road-Cd
            before the FAILURE: Hole-In-Tank!
6:30:42 (S) Sensing: gt2 with GT-Sensor...
6:33:57 Completed: (Drive gt1 from delta
                    to calypso on road-cd at 54)
6:35:23 (S) Sensing: calypso with City-Sensor...
6:37:21 (S) Sensing: c5-1 with ACT-Sensor...
7:08:47 (S) Sensing: c5-1 with ACT-Sensor...
7:09:11 (S) Sensing: road-ae with Road-Sensor...
7:10:14 (E) Bay-Bridge ATTACKED by T-Fal-1
             Status is now Closed.
7:15:57 (S) Sensing: gt1 with GT-Sensor...
```

Figure 8.5: Simulation History Snapshot

replace-fuel-tank task (Figure 8.4) are not satisfied. Hence, the planning agent must be notified since the REA does not possess any other applicable knowledge concerning the how to address the failure.

```
0~05:40:48...Considering task DRIVE-1 with status R
             of (DRIVE GT2 BARNACLE DELTA)
             DRIVE-1 is a NETWORK task.
             The network is
              ((GO-LOCATION GT2 DELTA BARNACLE ROAD-CD 46)
               (CITY-SENSOR BARNACLE))
0~05:40:54...Creating task GO-LOCATION-1 of DRIVE-1
0~05:40:57...Suspending task DRIVE-1
```

```
              of (DRIVE GT2 BARNACLE DELTA)
 0~05:41:05...Considering task GO-LOCATION-1 with status R of DRIVE-1
              GO-LOCATION-1 is a PRIMITIVE task.
 0~05:41:09...Dispatching task GO-LOCATION-1 of DRIVE-1
 0~05:41:23...(DRIVE GT2 DELTA BARNACLE ROAD-CD 46) dispatched.
 0~06:08:24...Performing active behavior UPDATE-ROAD-INFORMATION.

:KS-FAILURE0 **********************************
 KS-FAILURE: Receiving failure event...
 The reason for failure was: HOLE-IN-TANK
 Checking for specialist to handle HOLE-IN-TANK
 No specialist found.  Checking general knowledge...
<== Posting HOLE-IN-TANK for :NOTIFY ==>
:KS-FAILURE0 ******** END OF KS-FAILURE *******

:KS-NOTIFY0 **********************************
 Receiving UNABLE-TO-SATISFY-PRE-CONDITIONS event...
 0~06:22:06...********* PLAN FAILURE *********
              Packaging and Failing to Planning Agent...
              Type of Failure: UNABLE-TO-SATISFY-PRE-CONDITIONS
              Reason for failure: HOLE-IN-TANK
              Could not satisfy preconditions of: REPLACE-FUEL-TANK
:KS-NOTIFY0 ******** END OF KS-NOTIFY ********
```

The REA must therefore notify the planning agent that a failure has occurred. It does this via the Notify capability. The Failure capability determines that there is knowledge to address the failure, but that the conditions of applying that knowledge are not satisfied. It therefore posts an intention for the Notify capability to notify the planning agent that a failure has occurred and the reason it could not be addressed by the REA was because it was unable to satisfy the pre-conditions for applying the repair. The Notify capability gathers information related to the failure and plan status, and communicates that to the planning agent in the form of an IACL execution-failure message. The information in this message contains the reason for the failure in the environment, the reason the REA could not address the failure, the knowledge that it tried to apply to address the failure, the related causal structure, and a tag identifying the plan.

```
 0~06:34:49...Considering task DRIVE-1 with status P
              of (DRIVE GT1 CALYPSO DELTA)
 0~06:34:52...Network of DRIVE-1 is:
              (#<SCHEMA NODE-55.0 CITY-SENSOR>)
 0~06:34:55...Eligible procedures are: (#<PROCEDURE CITY-SENSOR-1>)
```

```
0~06:34:57...Creating task CITY-SENSOR-1 of DRIVE-1
0~06:35:01...Suspending task DRIVE-1
             of (DRIVE GT1 CALYPSO DELTA)
0~06:35:06...Considering task CITY-SENSOR-1 with status R of DRIVE-1
             CITY-SENSOR-1 is a PRIMITIVE task.
0~06:35:10...Dispatching task CITY-SENSOR-1 of DRIVE-1
0~06:35:21...(CITY-SENSOR CALYPSO) dispatched.
0~06:35:28...Processing FAILED TD...
0~06:36:03...Considering task DRIVE-1 with status P
             of (DRIVE GT1 CALYPSO DELTA)
0~06:36:06...Network of DRIVE-1 is empty.
0~06:36:08...DRIVE-1 has completed.
0~07:08:54...Performing active behavior UPDATE-ROAD-INFORMATION.
```

Execution of the plan continues to the point where the next high-level task is to be executed, and then halts at time 6 hours 36 minutes. That is, the gt1 transport continues on to Calypso. The REA issues a city-sensor request for Calypso since it is a subtask of the drive task. Upon completion of the drive task, the next task to be executed is the high-level load task. However, since the REA knows that a failure related to this plan has occurred, the load task is not carried out and execution of this particular plan halts.

```
(:EXECUTION-FAILURE
 (points-of-failure
    (NODE NODE-6-1)
    (NODE NODE-5-1))
 (gost-tags
   (TAG CSTR-13))
 (failure UNABLE-TO-SATISFY-PRE-CONDITIONS)
 (resource GT2)
 (reason WHOLE-IN-TANK))
```

Figure 8.6: IACL message for whole in tank failure

The points-of-failure information provided to the planning agent as a result of the failure allows it to determine which tasks were executing at the time of the failure (Figure 8.6). By halting execution of the plan at the next high-level task (i.e., continuing execution to that point) gives the planning agent an idea of what situation it is to plan for. For this example, it knows that it has to repair the gt2 resource (or use another resource) and continue to Barnacle to pick up the waiting nationals. It also

can determine that the gtl resource is in Calypso since that was the result of the last high-level task executed.

8.2.3 Innate Behavior

In Section 6.5.1 it was stated that if the REA was able to act without a Task Directive and (1) perform some internally generated task, and (2) react to some external stimulus, then it could be said to possess the innate behavior characteristic.

We have already seen these two cases demonstrated in other examples in this section. For example, when discussing guaranteed response we saw that the REA can perform tasks using its active behavior mechanism that occur whether the REA has a Task Directive or not. This could have easily been a passive behavior triggered by some change in a World Model value that required the REA to perform some internal maintenance.

In reacting to an external stimulus, we saw when discussing failure recovery that with the REA possessing a passive behavior and specialist knowledge that it was able to detect and address some change in the environment. This ability is independent of having a Task Directive. Thus, we have satisfied the criteria for the innate behavior characteristic.

8.2.4 Asynchronous Events

Actually showing the fact that the design allows for asynchronous events is difficult. We will attempt to show this by examining an execution snapshot for the situation where the REA is simultaneously receiving a sensor report from the environment and receiving an IACL message from the planning agent.

```
 0~01:00:03...Performing active behavior UPDATE-ROAD-INFORMATION.
 0~01:00:03...REA receiving IACL message: :STATUS
**********************************
KS-NO-CAPABILITY: Sending message to LEFTOut...
Unknown capability: :STATUS
**********************************
 0~01:00:04...Receiving event from environment...
```

Here we see that at 1 hour, 3 seconds the REA is simultaneously dispatching a sensor request and receiving an IACL status message. The REA sends an IACL no-capability message back to the planning agent since it does not know what a status message is, and at 1 hour, 4 seconds it receives the information from the sensor.

Another example of asynchronicity is the situation where the REA receives two IACL messages at the same time. Here the REA receives IACL messages of status and synthesize at time 19 seconds.

```
 0~00:00:19...REA receiving IACL message: :STATUS
 0~00:00:19...REA receiving IACL message: :SYNTHESIZE-NEW

####################################################
                Agenda (cycle:    2)


<AE-3    : 10 (NO-CAPABILITY (:STATUS)>
<AE-4    : 10 (NO-KNOWLEDGE (DRIVE-THE-THANG)>

--------------- Untriggered Entries ---------------

 No Untriggered entries...

####################################################

 0~00:00:19...KS-NO-CAPABILITY: Sending message to LEFTOut...
              Unknown capability: :STATUS

 0~00:00:19...KS-NO-KNOWLEDGE: Sending message to planning agent...
              Unknown knowledge: DRIVE-THE-THANG
```

In cycle 2, as the messages are received, the Guard (Section 4.3.2) notices that the REA does not have the capability to process a status message so it posts an intention for the No-Capability capability. At the same time in analyzing the synthesize message the Guard notices a task named drive-the-thang will not be understood by the REA so it posts an intention of the No-Knowledge capability. We then see the messages from both of the capabilities as the REA's intentions are processed.

These examples show that the REA does indeed possess the characteristic of handling asynchronous events.

8.2.5 Weighing Alternatives

The criteria for possessing the characteristic of weighing alternatives, according to Section 6.5.1, are (1) selecting between several procedures, (2) selection between Domain Data Objects (DDOs), and (3) selection of a Task Directive when more than one are present.

How the weighing of alternatives is performed was discussed in Section 4.4.3, and it is there where we show how the design allows the REA to select between several procedures of a particular task and between Task Directives. The only criteria that we can actually present as an execution example of this is the selection between DDOs.

Selection between Domain Data Objects

In this example, we examine the process of selecting between Domain Data Objects when several are eligible to satisfy a task specification. We join the execution where the Execute capability is selecting DDOs for each task in the network of the fly-transport-1 task.

For each of the subtasks in this network there are only one DDO which is available to select, except for the refuel subtask. Thus, we will concentrate on that task. The first step is to find all of the DDOs that match the pattern (refuel ?res), where ?res is the resource c5-1. It finds that the DDOs refuel-plane, refuel-helicopter, and refuel-gt all satisfy the pattern. Since multiple DDOs can potentially satisfy the subtask refuel, the REA must weigh the alternatives and select the best candidate given the information available at this point in the execution.

```
0~00:00:14...Considering task FLY-TRANSPORT-1 with status R
             of (FLY-TRANSPORT C5-1 DELTA CITY-K)
             FLY-TRANSPORT-1 is a NETWORK task.
             The network is ((LOAD C5-1 CITY-K GTS-ONLY)
                             (FLY-PLANE-TO-DEST C5-1 CITY-K DELTA)
                             (ACT-SENSOR C5-1) (UNLOAD C5-1 DELTA)
                             (REFUEL C5-1))
             .
             .
             .
:BUILD In find-DDO-with-pattern: (REFUEL C5-1)
```

```
:BUILD DDO-match : ((?RES . C5-1))
:BUILD Exiting Find-DDO-with-pattern having found:
      (#<DOMAIN-DATA REFUEL-PLANE>
       #<DOMAIN-DATA REFUEL-HELICOPTER>
       #<DOMAIN-DATA REFUEL-GT>)
:WA Selecting a DDO from: (#<DOMAIN-DATA REFUEL-PLANE>
                           #<DOMAIN-DATA REFUEL-HELICOPTER>
                           #<DOMAIN-DATA REFUEL-GT>)
:WA ORDER-OPTIONS: ((#<DOMAIN-DATA REFUEL-PLANE> 0 2 3 2)
                    (#<DOMAIN-DATA REFUEL-HELICOPTER> 0 2 3 1)
                    (#<DOMAIN-DATA REFUEL-GT> 0 2 3 2))
:WA Recommending #<DOMAIN-DATA REFUEL-HELICOPTER> from
 (#<DOMAIN-DATA REFUEL-HELICOPTER> #<DOMAIN-DATA REFUEL-PLANE>
  #<DOMAIN-DATA REFUEL-GT>)
:BUILD Checking conditions for #<DOMAIN-DATA REFUEL-HELICOPTER>
:BUILD Checking conditions for #<DOMAIN-DATA REFUEL-PLANE>
:BUILD Checking conditions for #<DOMAIN-DATA REFUEL-GT>
:BUILD DDOs satisfying conditions are (#<DOMAIN-DATA REFUEL-PLANE>)
:BUILD Synthesizing procedure objects of #<DOMAIN-DATA REFUEL-PLANE>
:BUILD Synthesizing the REFUEL-PLANE-1 PRIMITIVE procedure
:BUILD Synthesizing the REFUEL-PLANE-2 PRIMITIVE procedure
:BUILD Synthesizing the REFUEL-PLANE-3 PRIMITIVE procedure
```

The selection heuristics, discussed in Section 4.4.3, are used to order the candidates. These candidates are then analyzed and those not satisfying their applicability conditions are eliminated. In this example, we see that since the resource we are to refuel is an air cargo transport the only eligible candidate after considering the conditions of each DDO is refuel-plane. Thus, refuel-plane is selected and each of its procedures are synthesized.

The REA design does allow alternatives to be weighed. Here we have seen how it is done for the selection between multiple DDOs, and we have discussed previously how it is done for Task Directives and procedures. Therefore, the REA does possess the characteristic of weighing alternatives.

8.2.6 Change of Focus

In order to satisfy the criteria for the change of focus characteristic, we need to show that a change of processing focus occurs when (1) an event is detected by a behavior, (2) a failure is detected, and (3) when a protection monitor is violated.

We have already seen each of these demonstrated in Section 8.2.2. Criteria (1) was shown in discussing how the REA is able to recover from a failure via the behavior mechanism. The REA was processing the Task Directive for the NEO when the passive behavior plane-tire-or-mech-problem detected a blown tire on the B707 air passenger transport. The REA's focus changed from processing the Task Directive to addressing the blown tire with the Plane-Failure capability. Criteria (2) was shown when discussing how the REA is able to recover from an exogenous event. In this example the REA changed its focus from processing the Task Directive to addressing the broken fan belt on the gt1 ground transport with the general knowledge of the Failure capability. Finally, criteria (3) was shown when the detection of a potential failure by a protection monitor caused the REA to change its focus from the Task Directive to that of notifying the planning agent of the violation.

Thus, we have shown how the design satisfies the change of focus characteristic.

8.2.7 Predictability

The most difficult characteristic to demonstrate is that of predictability. In fact, that was one of the most difficult aspects to identify when characterizing the other systems in Chapter 2. However, we have seen that the behavior of the REA is predictable for particular situations in the Pacifica Domain. When a failure occurs in the environment that is detectable by the REA, then either a task is dispatched to address it or assistance is requested from the planning agent. When the REA receives an IACL message, it either processes it or notifies the planning agent why it was unable to process it.

In Section 6.5.2 we identified two primary types of factors that affect situated agents and discribed a set of expectations related to how we expected the agent to behave. Since each of these expectations were met in tests with the Pacifica Simulator, we can say that we believe the design satisfies the criteria for the predictability characteristic.

8.2.8 Temporal Reasoning

The definition of the temporal reasoning characteristic is rather simplistic and, as such, open to many interpretations. The intention was to imply that a particular system

should be able to represent temporal relations between the tasks it intends to execute. Again however, we could have one system able to represent only an after relation, and another able to represent the thirteen Allen relations [Allen 81] and under this definition both systems would be said to possess this characteristic. This is a shortcoming of several of the definitions in the characterization which will be discussed in the next chapter.

In this section we will see how the REA design is able to represent the temporal relations of before and after with respect to subtasks of a network procedure. In the design, temporal constraints are not intended to be hard[2], but rather a guide on how to temporally order subtasks in relation to one another.

We begin with an execution example in order to discuss the after temporal relation.

```
'(LOAD-2 network
  (context (and (o-type '?res 'air-cargo-transport)
                (at '?res '?location)))
  (network ((tn1 (gt-sensor gt1) ())
            (tn2 (gt-sensor gt2) ())
            (tn3 (act-sensor ?res) ())
            (tn4 (load-cargo-plane ?res ?location ?cargo)
                 ())))
  (ordering ((tn1 nil (tn4))
             (tn2 nil (tn4))
             (tn3 nil (tn4))
             (tn4 (tn1 tn2 tn3) nil)))
  (t-cons ((tn1 (AFTER tn3 120 180))
           (tn2 (AFTER tn1 60 120))
           (tn3 nil)
           (tn4 nil))))
```

Figure 8.7: Load-2 Procedure of the Load DDO

Temporal After Relation

In this example, we consider the execution of the load subtask of the fly-transport-1 task. We join execution after the Task Directive has been synthesized where the

[2] They may be more commonly referred to as preferences than constraints.

Execute capability is beginning to process the load-2 task (Figure 8.7).

```
:KS-EXECUTE0 **********************************
 0~00:00:11...Considering task LOAD-2 with status R of FLY-TRANSPORT-1
              LOAD-2 is a NETWORK task.
              The network is ((GT-SENSOR GT1)
                              (GT-SENSOR GT2)
                              (ACT-SENSOR C5-1)
                              (LOAD-CARGO-PLANE C5-1 CITY-K GTS-ONLY))
              Tasks eligible before checking temporal cons:
              (#<PROCEDURE GT-SENSOR-1> #<PROCEDURE GT-SENSOR-1>
               #<PROCEDURE ACT-SENSOR-1>)
              Tasks eligible after checking temporal cons:
              (#<PROCEDURE ACT-SENSOR-1>)
 0~00:00:11...Creating task ACT-SENSOR-1 of LOAD-2
 0~00:00:11...Suspending task LOAD-2
              of FLY-TRANSPORT-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******

 0~00:00:13...(AIR-CARGO-TRANSPORT-SENSOR C5-1) dispatched.
```

```
       0:13 (S) Sensing: c5-1 with ACT-Sensor...
       0:45 (E) Bay-Bridge ATTACKED by T-Fal-1
                 Status is now Closed.
       0:45 (E) Climate in Calypso is now Stormy.
       1:02 (E) Volcanic activity...Road-BD is now closed.
       2:00 (E) Bay-Bridge ATTACKED by T-Fal-1
                 Status is now Closed.
       2:15 (S) Sensing: gt1 with GT-Sensor...
       2:47 (E) Volcanic activity...Road-BD is now closed.
       3:18 (S) Sensing: gt2 with GT-Sensor...
       3:19 Started: (Load-Cargo c5-1 at city-k with gts-only)
```

Figure 8.8: Simulation History Snapshot

The Execute capability determines which tasks are eligible for execution according to load-2's ordering constraints. It then looks to see if there are any temporal constraints that should be applied. If temporal constraints do exist then these take precedence over the ordering constraints. In this case (Figure 8.7), we see that according to the ordering constraints tn1, tn2, and tn3 are all eligible to execute. However, the temporal constraints say that tn1 is to execute 2-3 minutes after tn3, and tn2 is to execute 1-2

minutes after tn1. Therefore, since tn3 has no temporal constraints and it is eligible to execute, it is dispatched.

When the act-sensor-1 subtask completes (Figure 8.8) and its effects are asserted into the REA's World Model, the load-2 task is again triggered and processed by the Execute capability. Information regarding the finishing time of the act-sensor-1 task is stored in the load-2 procedure object, and any temporal constraints for the act-sensor-1 task are deleted. This allows the Execute capability to determine which constraints are still valid (i.e., must be considered) and to be able to delay further execution at appropriate intervals between tasks.

To get a better understanding of how the temporal reasoning mechanism works, we examine the internals of the load-2 procedure object at this point in the execution (Figure 8.9). Here we see that the start-time slot holds a timestamp of 11 has the beginning of execution for the load-2 task.

```
#<PROCEDURE LOAD-2>
 NAME              LOAD-2
 ITYPE             LOAD
 E-STATUS          P
 OWNER             #<TASK-DIRECTIVE T-DIRECTIVE-1>
 PARENT            #<PROCEDURE FLY-TRANSPORT-1>
 SERVICING         #<SCHEMA NODE-1.0 FLY-TRANSPORT>
 REFNODE           PROCNET-21
 P-TYPE            NETWORK
 EXECUTED          (PROCNET-34)
 START-TIME        11
 TEMPORAL-INFO     ((PROCNET-34 14))
 CONSTRAINTS       ((PROCNET-32 (AFTER PROCNET-34 120 180))
                    (PROCNET-33 (AFTER PROCNET-32 60 120))
                    (PROCNET-35 NIL))
```

Figure 8.9: Load-2 task beginning execution

At time 14, the tn3 (i.e., act-sensor-1) subtask completed[3]. This information is stored in the temporal-info and executed slots. We also see that the temporal constraints have been updated to reflect the fact that the constraints of the tn3 task are no longer valid.

```
:KS-EXECUTEO **********************************
 0~00:00:14...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~00:00:14...Network of LOAD-2 is:
              (#<SCHEMA NODE-27.0 GT-SENSOR>
               #<SCHEMA NODE-28.0 GT-SENSOR>
               #<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
 0~00:00:14...Checking Temporal Constraints of:
              #<PROCEDURE LOAD-2>
              Eligible tasks before checking constraints:
              (#<PROCEDURE GT-SENSOR-1> #<PROCEDURE GT-SENSOR-1>)
              Temporal constraints are:
              ((PROCNET-32 (AFTER PROCNET-34 120 180)))
 0~00:00:14...Found one valid temporal constraint.
              Delaying #<PROCEDURE LOAD-2> for 120 seconds
:KS-EXECUTEO ****** END OF KS-EXECUTE ******
```

When the Execute capability again examines the load-2 task it determines that subtasks tn1 and tn2 are eligible to execute. Upon considering the temporal constraints, it determines that only tn1 is eligible to execute, but that it is constrained to execute 2-3 minutes after tn3. Therefore, Execute delays the further execution of the load-2 task by 120 seconds from the time the tn3 task completed. In other words, the execution of the load-2 task will not be considered again for 120 seconds.

```
:KS-EXECUTEO **********************************
 0~00:02:14...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~00:02:14...Network of LOAD-2 is:
              (#<SCHEMA NODE-27.0 GT-SENSOR>
               #<SCHEMA NODE-28.0 GT-SENSOR>
               #<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
 0~00:02:14...Checking Temporal Constraints of:
              #<PROCEDURE LOAD-2>
              Eligible tasks before checking constraints:
```

[3] When the information from a Domain Data Object is used by the REA, the references to subtasks are made unique. That is, references to tn1, tn2, etc. are not unique between tasks, so the REA makes them unique by assigning procnet numbers. In this example, procnet-32 corresponds to tn1, 33 to tn2, 34 to tn3, and 35 to tn4. In fact, since the load-2 task itself is a subtask of fly-transport-1, it has a reference of procnet-21.

```
              (#<PROCEDURE GT-SENSOR-1> #<PROCEDURE GT-SENSOR-1>)
              Temporal constraints are:
              ((PROCNET-32 (AFTER PROCNET-34 120 180)))
 0~00:02:15...Found one valid temporal constraint.
 0~00:02:15...Eligible procedures after checking are:
              (#<PROCEDURE GT-SENSOR-1>)
 0~00:02:15...Creating task GT-SENSOR-1 of LOAD-2
 0~00:02:15...Suspending task LOAD-2
              of FLY-TRANSPORT-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******

:KS-EXECUTE0 **********************************
 0~00:02:15...Considering task GT-SENSOR-1 with status R of LOAD-2
              GT-SENSOR-1 is a PRIMITIVE task.
 0~00:02:15...Dispatching task GT-SENSOR-1 of LOAD-2
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******

 0~00:02:15...(GROUND-TRANSPORT-SENSOR GT1) dispatched.
```

After the 120 second delay, the load-2 task is again considered by the Execute capability. Again the constraints are examined, and this time the tn1 subtask is found to be eligible to execute. The tn1 subtask is then dispatched.

```
EXECUTED          (PROCNET-32 PROCNET-34)
START-TIME        11
TEMPORAL-INFO     ((PROCNET-32 136) (PROCNET-34 14))
CONSTRAINTS       ((PROCNET-33 (AFTER PROCNET-32 60 120))
                   (PROCNET-35 NIL))
```

Figure 8.10: Load-2 task information during execution

When the tn1 (i.e., gt-sensor-1) subtask completes and its effects asserted into the REA's World Model, the Execute capability again begins processing of the load-2 task. Once again the information in the load-2 procedure object is updated to reflect the fact that a subtask has completed (Figure 8.10).

```
:KS-EXECUTE0 **********************************
 0~00:02:16...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~00:02:16...Network of LOAD-2 is:
              (#<SCHEMA NODE-28.0 GT-SENSOR>
               #<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
 0~00:02:16...Checking Temporal Constraints of:
```

```
              #<PROCEDURE LOAD-2>
              Eligible tasks before checking constraints:
              (#<PROCEDURE GT-SENSOR-1>)
              Temporal constraints are:
              ((PROCNET-33 (AFTER PROCNET-32 60 120))
0~00:02:16...Found one valid temporal constraint.
              Delaying #<PROCEDURE LOAD-2> for 60 seconds
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******
```

Execute again checks the constraints and determines that tn2 is now eligible to be executed, but that cannot occur until 1-2 minutes after the completion of the tn1 task. Therefore, further processing of the load-2 task is delayed for 60 seconds.

After 60 seconds, Execute processes the load-2 task again and this time it dispatches the tn2 subtask.

```
 0~00:03:17...(GROUND-TRANSPORT-SENSOR GT2) dispatched.

:KS-EXECUTE0 **********************************
 0~00:03:18...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~00:03:18...Network of LOAD-2 is:
              (#<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
 0~00:03:18...Checking Temporal Constraints of:
              #<PROCEDURE LOAD-2>
              Eligible tasks before checking constraints:
              (#<PROCEDURE LOAD-CARGO-PLANE-1>)
              Temporal constraints are: NIL
              CONSTRAINTS: ((PROCNET-35 NIL))
              Remaining temporal constraints are ignorable!
 0~00:03:19...Creating task LOAD-CARGO-PLANE-1 of LOAD-2
 0~00:03:19...Suspending task LOAD-2
              of FLY-TRANSPORT-1
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******

:KS-EXECUTE0 **********************************
 0~00:03:19...Considering task LOAD-CARGO-PLANE-1 with status R
              of LOAD-2
              LOAD-CARGO-PLANE-1 is a PRIMITIVE task.
 0~00:03:19...Dispatching task LOAD-CARGO-PLANE-1 of LOAD-2
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******

 0~00:03:19...(LOAD-CARGO C5-1 CITY-K GTS-ONLY) dispatched.
```

At time 3 minutes, 18 seconds, we see that tn2 has completed and that Execute is processing the load-2 task. This time however, the only remaining temporal constraint

(i.e., for tn4) is nil so, it can be ignored and execution of the tn4 subtask begins. Here we have seen the temporal reasoning capability of the REA for temporally ordering subtasks in relation to one another.

Temporal Before Relation

In this example, we will again consider the execution of the load-2 task, but this time we will see how the after relation can be used to specify a before constraint between two subtasks. The temporal constraints for this example are shown in Figure 8.11. Here we see that the tn2 task is temporally constrained to executed before the tn1 task, and that both tn1 and tn2 are temporally constrained to execute after the tn3 task.

```
(t-cons ((tn1 (AFTER tn3 120 180))
         (tn2 (AFTER tn3 30 90))
         (tn3 nil)
         (tn4 nil))))
```

Figure 8.11: Load-2 Temporal Constraints

We examine the execution trace at the point in execution when the tn3 (i.e., act-sensor-1) subtask has just completed and its effects asserted into the REA's World Model. The simulation history for this example can be seen in Figure 8.12.

```
0:27 (S) Sensing: c5-1 with ACT-Sensor...
0:59 (E) Bay-Bridge ATTACKED by T-Fal-1
         Status is now Closed.
0:59 (E) Climate in Calypso is now Stormy.
0:59 (S) Sensing: gt2 with GT-Sensor...
1:16 (E) Volcanic activity...Road-BD is now closed.
1:51 (E) Bay-Bridge ATTACKED by T-Fal-1
         Status is now Closed.
2:29 (S) Sensing: gt1 with GT-Sensor...
2:31 Started: (Load-Cargo c5-1 at city-k with gts-only)
```

Figure 8.12: Simulation History Snapshot

```
:KS-EXECUTE0 **********************************
 0~00:00:28...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~00:00:28...Network of LOAD-2 is:
              (#<SCHEMA NODE-27.0 GT-SENSOR>
               #<SCHEMA NODE-28.0 GT-SENSOR>
               #<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
 0~00:00:28...Checking Temporal Constraints of:
              #<PROCEDURE LOAD-2>
              Eligible tasks before checking constraints:
              (#<PROCEDURE GT-SENSOR-1> #<PROCEDURE GT-SENSOR-1>)
              Temporal constraints are:
              ((PROCNET-33 (AFTER PROCNET-34 30 90))
               (PROCNET-32 (AFTER PROCNET-34 120 180)))
 0~00:00:28...Multiple temporal constraints found.  Checking each...
              Found multiple valid temporal constraints.
              Choosing the one with the earliest start time...
              Delaying #<PROCEDURE LOAD-2> for 30 seconds
:KS-EXECUTE0 ****** END OF KS-EXECUTE ******
```

When the Execute capability examines the temporal constraints it determines that multiple constraints are applicable. In this case, since the tn3 (i.e., procnet-34) subtask has completed and the constraints for tn1 and tn2 are in relation to tn3, Execute must select a constraint to determine how long to delay the execution of the load-2 task. It does this according to the earliest start time for the temporal window of the constraints. Here, it chooses to delay the execution of load-2 by 30 seconds according to the tn2 constraint.

```
:KS-EXECUTE0 **********************************
 0~00:00:58...Considering task LOAD-2 with status P of FLY-TRANSPORT-1
 0~00:00:58...Network of LOAD-2 is:
              (#<SCHEMA NODE-27.0 GT-SENSOR>
               #<SCHEMA NODE-28.0 GT-SENSOR>
               #<SCHEMA NODE-30.0 LOAD-CARGO-PLANE>)
 0~00:00:59...Checking Temporal Constraints of:
              #<PROCEDURE LOAD-2>
              Eligible tasks before checking constraints:
              (#<PROCEDURE GT-SENSOR-1> #<PROCEDURE GT-SENSOR-1>)
              Temporal constraints are:
              ((PROCNET-33 (AFTER PROCNET-34 30 90))
               (PROCNET-32 (AFTER PROCNET-34 120 180)))
 0~00:00:59...Multiple temporal constraints found.  Checking each...
              Constraint within window.  Executing...
 0~00:00:59...Eligible procedures after checking are:
              (#<PROCEDURE GT-SENSOR-1>)
```

```
 0~00:00:59...Creating task GT-SENSOR-1 of LOAD-2
 0~00:00:59...Suspending task LOAD-2
              of FLY-TRANSPORT-1
:KS-EXECUTEO ****** END OF KS-EXECUTE ******

 0~00:00:59...(GROUND-TRANSPORT-SENSOR GT2) dispatched.
```

Once the 30 seconds has expired, the Execute capability again considers the constraints and now finds that tn2 is within its window of execution and dispatches that subtask. Upon completion of the tn2 (i.e., gt-sensor-1) subtask, Execute again considers the constraints and now delays the further execution of load-2 for 88 seconds. That is, it delays the execution 120 seconds from the time tn3 completed. When 88 seconds have passed, Execute dispatches the tn1 task.

As in the previous example, when Execute again considers the constraints it finds that there are no more constraints so it adheres to the ordering constraints. This causes the last subtask, tn4, to be executed.

Thus, by specifying two subtasks in relation to another (i.e., tn1 and tn2 in relation to tn3) we can establish a temporal before relation on those tasks. In this example, tn2 was executed before tn1.

8.3 MAD

8.3.1 Explanation

Fortunately, or perhaps unfortunately depending upon your point of view, the architecture performed as expected. Thus, there is nothing to describe in regards to why something happened and how it could be corrected. Perhaps since we discuss some implementation shortcomings in Section 9.3 those could be considered to satisfy the explanation criterion of the MAD methodology.

8.3.2 Generalization

The REA was designed to be a general purpose architecture for monitoring the execution of plans, and as such, it is hoped that the types of environments to which such an

architecture could be applied to is limitless. However, we have only shown its applicability to a command and control type environment. In order to apply the architecture to new environments one would have to provide (1) Task Behavior Language specifications appropriate to the tasks to be executed in the environment, (2) capabilities and general failure handling knowledge for addressing failures from the environment, and (3) domain knowledge (e.g., knowledge of available sensors, resources, and general environmental information) for the environment in which the REA is to be situated.

8.4 Chapter Summary

We have considered how the REA design is able to satisfy the criteria set forth in Chapter 6 and demonstrate each of the characteristics of the characterization from Chapter 2. The detailed examples have offered evidence of how the design provides for guaranteed response, failure recovery, innate behavior, asynchronous event handling, the weighing of alternatives, the change of processing focus, predictability, and temporal reasoning. In addition, we have seen in these examples other features of the design such as protection monitors that allow the REA to predict potential failures, active sensing, and communication adaptability and enhancement of behaviors, capability, and knowledge.

In the next chapter, we will consider how the characterization from Chapter 2 can be improved in the light of implementing those characteristics for the design presented here, discuss areas where the design can be improved, and summarize this research.

Chapter 9

Conclusions and Future Research

For many years researchers have either adopted the approach that better plan generation is the key to intelligent execution, or the approach that less deliberation and more reacting is how to achieve intelligent execution. What has become clear is that planning and execution are complementary processes and neither can be successful to the total exclusion of the other. This realization is what we have examined in this research—how best to utilize the benefits of plan generation and execution to achieve more intelligent execution.

We chose to use a top–down approach that involved characterizing the behavior we wished our system to possess, designing an architecture that could integrate those characteristics in a modular fashion, and developing additional features to show how more effective execution monitoring could be achieved. We chose to separate plan generation and execution into two processes to avoid having to trade off deliberation time versus reaction time. To utilize the benefits of the separate processes we thought it best to develop a flexible communication language that would allow the two processes to share different types of information in order to adapt to the environment.

The preceding chapters have discussed this design approach, introduced an architecture that integrates desired characteristics for rational behavior in a competent manner, and discussed how the implementation allows our agent to behave in the complex and dynamic environment of Pacifica. In this final chapter, we reconsider the characterization of rational behavior in order to complete the design cycle and extend that character-

ization to include important aspects identified in this research. We then discuss the future research that needs to be addressed as a result of this research. This includes general topics related to the field as a whole, and topics specifically related to the design approach.

9.1 Characterization of Rational Behavior

The purpose of the characterization of rational behavior presented in Chapter 2 was to provide a common frame of reference to discuss various architectural features. This was in order to relate the internal characteristics of various architectures to externally observable properties of instances of those architectures [Drummond & Kaelbling 90]. When we presented the characterization, we validated it by comparatively evaluating how a representative sample of the main contributing systems in the areas of reactive execution and integrated architectures could be rated using such a characterization.

As a result of the comparative evaluation and of trying to design an agent which incorporated each of the characteristics into a new style of architecture, we determined that our definitions were not detailed enough to isolate the particular behavior we desired. What is needed is a clear specification of the types of observable behavior that a particular characteristic may exhibit so that we can quantitatively measure the degree to which a system possesses that characteristic and indentify the best approaches. Thus, enabling us to design better systems.

In this section, we complete one cycle of the top–down design approach by again considering the characterization. We intend to apply the understanding we have gained through this research to developing an enhanced characterization that can be utilized in future designs of rational execution agents. We begin by extending the definitions in the original characterization to detail the particular behaviors that together compose a characteristic. We then extend the characterization itself to include other aspects of rational behavior that have been identified as significant during this research.

The original characterization identified eight characteristics that together provided a guideline for developing systems that could behave rationally. These characteristics were guaranteed response, failure recovery, innate behavior, asynchronous events,

weighing alternatives, change of focus, predictability, and temporal reasoning. We begin in the next section by examining the definitions of each of these characteristics.

9.1.1 Enhanced Characterization

Guaranteed response was defined as the ability to guarantee some kind of response by the time a response is required. However, a time interval of zero would be truly impossible—there is *some* computation required, no matter how trivial, that must go on before an agent decides what to do. Hence, we relaxed this definition to allow for random responses, default responses, and bounded responses. Nonetheless, because doing nothing is a valid response (and very often the right response) we would also need to add that to the definition, thus yielding the undesired effect of making every architecture possess the guaranteed response characteristic. The original idea behind this characteristic was to be able to guarantee a response to an internal or external event and continue to function whether the agent was able to handle the event or not. The definition became murky when we tried to get it to double as a criterion for providing a response that was timely based upon the environment in which the agent was situated. In an attempt to clarify the definition we will confine it to mean the ability of the agent to provide a response to an event in bounded time as defined according to the reasonable temporal horizon for the environment in which the agent is situated. The response can be default, random, or monotonically improving in time towards the best possible response. We will make the requirement for continued operation a separate characteristic since it lends itself more to the issue of robustness than a particular type of response.

The failure recovery characteristic had the simplistic definition of the ability of the agent to continue to operate after a failure was detected either as the result of a task failure or an exogenous event. Though the continuous operation of an agent is a worthy goal, this definition has little, if anything, to do with failure recovery, and as such, has been the source of a great deal of ambiguity and confusion. What we need is a pragmatic definition of how failures could possibly be managed because we cannot in the foreseeable future, design the perfect omnipotent agent [Hallam 94]. Hence, we will concern ourselves here with whether the agent is able to recover from cognizant

(or anticipated failures) and whether it can gracefully handle unanticipated failures without catastrophic collapse. When we talk about extending the characterization in Section 9.2, we will address the specific aspects of this behavior which should be exhibitable by agents that possess this characteristic.

Innate behavior was another definition that did not expressly state its intent. The original definition was that the agent be able to act without an explicit plan directing it to act. Though this definition captured the essential concept we wanted, it failed to explicitly specify the other aspects of such behavior. The intention behind this characteristic is that an agent be provided with some means to be self–sustaining in the environment in which it is situated by possessing behaviors which would allow it to function competently until such time as it received a plan specifically directing it to act. In [Drummond & Bresina 90a] they talk of prior behavior competence (or behavioral constraints), [McDermott 92] talks about default reactive plans, and [Lyons & Hendriks 92] talk about an abstract plan. These systems capture the essence of the innate behavior characteristic.

The definition of the asynchronous events characteristic captures the intent of the characteristic. In the definition by [Laffey *et al.* 88] they include the statement, "[the agent] must also be capable of processing input according to importance, even if the processing of less important input must be interrupted or rescheduled". This tends to lead one to think about the way asynchronous events will be processed and begins to obscure the definition. Here we will only be concerned with whether the agent can accept events while processing or accepting events simultaneously from different sources. We will leave the issue of how the events are processed to the definition of the change of focus characteristic.

The definition of weighing alternatives fulfills its purpose as well. What we failed to do previously was to define the aspects of this behavior so we can identify this specific behavior in other systems. What we want to determine is whether an agent bases its decisions on present environmental context, heuristics about the domain, temporal deadlines, resource utilization levels, or some other set of factors.

Change of focus was defined as the ability of the agent to focus processing attention on important tasks even if the processing of less important tasks must be rescheduled

or aborted. This definition implicitly states that the agent must possess the ability to determine the importance of events and that it should process the most important tasks first. To strengthen the definition, we shall now make this latter implicit statement explicit in order to characterize the exact behavior we expect our systems to exhibit.

Characteristic	Internal Behavior
Guaranteed Response	Default Response Random Response Bounded Response
Failure Recovery	Unanticipated Failure Cognizant Failure
Innate Behavior	Behavioral Competence Stimulus–Response Network Active/Passive Behaviors
Async. Events	Input/Output During Processing Simultaneous Input/Output
Weighting of Alternatives	Contextually Heuristically Temporally (closest to deadline) Utilization of Resources
Change of Focus	Prioritized Processing
Predictability	Stimulus–Response Correlation Task Effect Predictability Temporal Horizon Predictability Series Response Correlation
Temporal Reasoning	Point/Interval Relations (Tasks) Point/Interval Relations (Subtasks) Point/Interval Relations (Plans)

Table 9.1: Enhanced Characterization

Predictability is the most ambiguous characteristic of our set. Do we mean predictability to imply a phenomenological model[1] where knowledge about future states can be mathematically determined? That is, action X always produces Y 97% of the time. Or

[1] This term is credited to Oliver Sparrow.

do we mean that predictability implies a chain-of-events model? Where given the chain of events: action A, then B, then C we can expect response G. The intended definition is effectively a combination of these two models. What we want to be able to define is that for any input we can determine the output, not that we can predict the order of processing or interaction involved when a series of inputs are received at one time. We simply want to be able to say that we can "predict" that we know what the system will do for any particular situation taken in isolation. This is of course not enough by itself to guarantee coherent behavior when the system is actually deployed. Therefore, we shall refer to this definition as *simple* predictability, and define the term *complex* predictability for systems that are able to (1) predict the compound interaction of two or more actions, or (2) mathematically predict the occurrence of a particular action.

Finally, we consider the definition of the temporal reasoning characteristic. Again the definition satisfactorily captures the general intent while not being specific enough to encompass all of the specific intended nuances. Let's add to the definition that the agent should comprehend point and interval temporal representations, clipping, persistence, and all with respect to tasks, subtasks, and high–level plans. We want to capture the fact that the agent is completely able to reason about the aspects of time. This definition may now be too strict since temporal reasoning to this degree might not be required in all domains to achieve rational behavior. Therefore, we shall define this characteristic as the ability of a system to sufficiently represent temporal constraint information such that temporal relations, preferences, and constraints can be made with respect to tasks, subtasks, and high–level plans. If a particular system then chooses to include more advanced temporal reasoning mechanisms then so be it, as long as, this definition can be satisfied.

We summarize the enhanced characterization of rational behavior in Table 9.1.

9.2 Extending the Characterization

We intend to extend the characterization in two areas which have received considerable attention in this research and in the AI literature as of late, and in one area we alluded to in the previous section. These are the characteristics of adaptability, failure

management, and continuous operation respectively.

Considering the research presented in this dissertation and that undertaken by [Drummond & Bresina 90a,McDermott 92] and [Lyons *et al.* 91] on incremental adaptation and behavior transformation we shall define a characteristic for adaptability. It is defined as the ability of the agent to dynamically modify its behavior, add new behaviors, add (or modify) domain knowledge, and add (or modify) procedural knowledge to address specific situations when directed to do so by a superior agent. This characteristic reflects the fact that as the environments in which we situate agents become more dynamic and complex we must have the ability to dynamically adapt the behavior of those systems to address new and novel situations not previously encountered or considered.

A characteristic involving failure management is a refinement of the failure recovery characteristic to achieve a pragmatic way of specifying that rational systems be able to address execution failure. The purpose is to identify some internal aspects of failure management that will allow us to design agents whose approach to failure management is pragmatic. Therefore, we desire that an agent possess the ability to recover from cognizant and unanticipated failures, actively monitor for protection interval violations (i.e., monitor the execution of the plan), repair execution failures by re–ordering, removing, or inserting tasks, and to utilize clean–up procedures when tasks fail.

In the previous section we stated that the ability of an agent to continue to operate was an important behavior, but that including it as part of the defining criteria for the guaranteed response or failure recovery characteristics was inappropriate. We shall then extend the characterization by adding a continuous operation characteristic. It is defined as the ability of the agent to continually operate until directed to cease operation or upon the occurrence of a catastrophic event. This definition is a re–statement of the continuous operation characteristic presented in the characterization of real–time knowledge–based systems by [Laffey *et al.* 88].

We summarize the extended and final characterization of rational behavior in Table 9.2.

The problem with arbitrarily adding more and more characteristics is that we end up including everything as necessary for rational behavior (and blurring the distinction

Characteristic	Internal Behavior
Guaranteed Response	Default Response Random Response Bounded Response
Failure Management	Recover from Cognizant Failure Recover from Unanticipated Failure Monitor Protection Intervals and Execution Repair Execution Failures Locally Utilize Clean–up Procedures on Failure
Innate Behavior	Behavioral Competence Stimulus–Response Network Active/Passive Behaviors
Async. Events	Input/Output During Processing Simultaneous Input/Output
Weighting of Alternatives	Contextually Heuristically Temporally (closest to deadline) Utilization of Resources
Change of Focus	Prioritized Processing
Predictability	Stimulus–Response Correlation Task Effect Predictability Temporal Horizon Predictability Series Response Correlation
Temporal Reasoning	Point/Interval Relations (Tasks) Point/Interval Relations (Subtasks) Point/Interval Relations (Plans)
Adaptability	Add/Modify Processing Behavior Add/Modify Domain Knowledge Add/Modify Procedure Knowledge
Continuous Operation	Operate Until Directed to Halt Operate Until Catastrophic Failure

Table 9.2: Extended Characterization

of sufficient for such behavior). For instance, we could include resource reasoning, qualitative reasoning, goal–directed behavior, and the ability to minimize actions not in service of higher level goals as additional characteristics (just to name a few). It does not seem unreasonable, but we must consider the problems that such additions have on the processing resources of an agent. This is a problem which is presented as an area where more research needs to be conducted.

9.3 Future Research

Research in new areas such as intelligent execution systems tends to raise as many questions as it answers. The research in this dissertation suggests further research in two primary areas: first, in a general fashion, to develop additional reasoning capabilities for execution systems; second, to make specific additions or modifications to the proposed system to address some of its limitations or inefficiencies.

9.3.1 General Research Topics

Due to the breadth of the problem studied in this research, some of the ideas originally planned for could not be adequately addressed. Thus, several topics are left as open research and deserve separate attention. These are described in this section.

First, facilities for full temporal reasoning need to be integrated into the REA. Originally, the Time Map Manager (TMM) software [Boddy 91,Hon92] developed by Honeywell was to meet this need. TMM is a nonmonotonic, deductive, temporal database management system that provides many mechanisms that would allow an agent to possess powerful reasoning mechanisms. The intent was to take advantage of TMM's rich representation of time and allow an agent to reason about time points, time intervals, persistence, clipping, and temporal projection. The theory is that with such mechanisms available an agent could reason about more of the consequences of its actions and potentially become more robust (i.e., reporting failure less frequently). Needless to say, we did not have the opportunity to explore this avenue of research. However, since no other known execution systems possess these temporal reasoning features it would be an interesting research exercise to see if in fact, the theory could be shown

to be true.

Resource reasoning capabilities should be part of any intelligent execution system. An agent should be able to reason about utilization levels of resources under its control and be able to schedule task execution to avoid resource contention. The simplest approach is to use semaphores, since these do not depend upon a representation of time. However, such an approach limits the depth of reasoning that can be achieved, and it not satisfactory for all types of domains [Miller 85]. The best approach would be to couple a resource reasoning facility with that of a full temporal reasoning facility (such as TMM) to reap the greatest benefit from both facilities. This is another interesting topic that has received little attention in the literature on intelligent execution systems. Perhaps the literature on scheduling systems would be a good place to start.

Another interesting avenue for research might be adding contingency planning capabilities to the execution system. In the case of the REA executing NEO plans in Pacifica, there were often periods where the REA was idle due to the extended duration of the tasks it was executing. We might be able to use this time to develop contingencies for currently executing tasks to better address failures should they occur. The ERE system [Drummond & Bresina 90a] uses temporal projections to find optimal solution paths, but this information can also be used to address failures. Another interesting idea along the same lines would be to explore the benefits of specifying plans which contain contingencies to the execution system.

When discussing execution systems people are usually concerned with monitoring the execution of tasks to make sure that preconditions are satisfied or that effects have been achieved. However, it might be interesting to monitor the progress of a task towards achieving its goal (e.g., effects). This idea was discussed with Mark Drummond where we talked about an agent having expectation models for the tasks it could execute or for particular tasks that had a high probability of failure. By possessing such models the agent would be able to compare its status with that of the expectation model for an executing task at prescribed intervals during execution to determine if progress was being made towards it goal. If the tasks varied some degree beyond a threshold from what was expected then the agent would then be able to intervene and hopefully, get the execution back on track. Hart at the University of Massachusetts did some work

along these lines called envelopes for the Phoenix system [Hart *et al.* 90], and Kohout at the University of Maryland also has ideas on this subject [Kohout 93] following from the Dynamic Reaction Model of [Sanborn & Hendler 88].

In Erann Gat's dissertation [Gat 91], he talks about the Wesson Oil problem. In this problem, a woman is frying chicken (in Wesson Oil) when one of her children suddenly falls down and has to be taken to hospital. However, before going she turns off the stove. When she later returns she resumes frying the chicken. This is an interesting behavior which he addresses with an "unwind–protect" feature. This unwind–protect defines a clean–up procedure that should be executed when a high priority task interrupts the execution of a lower priority task. His solution, admittedly, is only a partial solution since the selection of the clean–up procedure is context dependent. However, it is a serious problem which deserves further consideration, and could potentially provide an execution system with a great deal of robustness in the face of execution failure.

The purpose of this section is to identify areas where there is potential for someone to address open research issues in the area of execution system design. The problem is that if someone were to develop each of the ideas from this section, we could easily build so much reasoning power into the execution system that we would again have the problem of trading deliberation time for reaction time. Therefore, the most interesting research topic of all (in my opinion of course) would be to identify the trade–offs so future designers could use that knowledge. For example, consider resource reasoning. How much of a benefit would it actually be? If we were to spend time deliberating about resources that would avoid problems arising from resource contention later, would that be better than addressing the resource contentions as they occurred? These issues may not be addressable in a general way and may require significant domain knowledge.

9.3.2 Extending the REA Approach

First, and foremost, to extending the REA approach is to make the necessary modifications for controlling an autonomous robot or a robotic cell. This would involve developing the domain knowledge and a new Dispatch capability to interface to the hardware. This would lend credibility to the design and architecture, and identify further limitations to be addressed.

The active sensing mechanism needs to be modified in two significant ways. First, in the way that the Sensor capability determines if a particular sensor request should be dispatched to the environment. Presently, if the sensor request is for the same resource that the sensor was last used to gather information from then it is ignored[2]. This approach does not guarantee what it should. It guarantees that no two sensor requests will be made for the same resource, but what it should guarantee is that for N resources that the sensor requests would be $R_1, R_2, \ldots, R_N$ and when R_N was reached that the next request would start again with R_1. Second, we could be smarter about the number of active behaviors that are created for active sensing. At present, given the patterns (at gt1) and (res-status gt1) from two distinct causal structure records we would have two active behaviors created that both require sensing of the gt1 ground transport resource using the same sensor. The problem being that the gt-sensor gathers information for both the *at* and *res-status* attributes so having separate sensor requests is redundant and unnecessary overhead. What we should do is check, at the time we are creating active behaviors from the causal structure information, that we are not already requesting information from a particular sensor. We would however, have to concern ourselves with the fact that by not issuing the requests separately that when we no longer required *res-status* information after, say node-4, that we could still continue to gather location information until node-23.

Another area where we might be able to provide additional insight to the REA is by rating tasks as to the seriousness of failure. For example, if the airport was under attack when a plane was supposed to take–off, then the fact that we did not get clearance from the tower does not mean that the fly–transport task should fail. Also, the failure of a sensory task should not cause an entire plan to fail to execute, as is presently the case. We need some method of representing such information to the execution system so that it can make decisions regarding the severity of an execution failure. This would help to make the system more robust and reduce the frequency of the execution system having to request assistance.

A mechanism to allow the execution system to reason about the effects of the task

[2] This is true except when there is only one resource of a particular type that requires sensing. For example, if information is to be gathered from only one air cargo transport then each time the Sensor capability gets a sensor request for it, the request is made.

specifications in its domain knowledge would also help to make the system more robust in the face of failure when precondition failures occur. In the current system, if a ground transport resource is to drive from one location to another and its mechanical status is bad, then the drive task will fail because all of its preconditions have not been satisfied. This causes the REA to send an IACL execution–failure message to the planning agent. If the REA could reason about the effects of other tasks in its domain knowledge (which is already included in the TBL representation of tasks) it would find that if it executed the task fix-gt-mech then it could locally repair the problem and continue with normal execution. Intuitively, a mechanism such as this would be a good idea, but it could potentially cause problems later if the fix-gt-mech task required the use of a limited resource that, unbeknownst to the REA at the time, was required later in the plan. This highlights the problem with myopic decisions taken at execution time versus allowing the planning agent to consider global concerns. [Drummond & Levinson 92] have begun to look at how a planning agent can monotonically increase the effective performance of an execution system. What is not clear is whether the execution system can increase the effective performance of a planning agent by making local decisions. Perhaps there is a class of decisions that the execution system could make locally that would not be a detriment to some other concern later in the plan.

A limitation in regards to the communication abilities of the REA concerns the interaction of the Guard in the Communication Manager with message events. We discussed in Chapter 5 how the Guard analyzes the contents of messages to determine whether the REA will understand the information contained in the message and have the capabilities to process the message. The problem is that the design of the Guard only allows it to analyze the information contained in an IACL Synthesize message. There should be a means to dynamically specify that the Guard analyze other types of messages if need be. This is not a major research issue; however, if the REA is to be truly adaptable it should not have any limitations such as this one[3].

Lastly, another important area of future research involves how execution failures are managed by the REA. At the end of Chapter 7 we discussed some of the ways that

[3] In the general O–Plan architecture it is intended that this limitation be eliminated by allowing the Guard to be "programmed" by using the details of the installed knowledge sources given by a knowledge source formalism which has yet to be fully specified [Tate 94].

the failure management capabilities of the REA might be improved. We will not shed any more light on the subject here except to again emphasize the importance of such research. This is another area where significant gains in building execution systems could be made.

9.3.3 Real–Time Processing Limitation

Though we were not attempting to design a new control regime in this research we have explored some new territory in trying to apply an existing control approach within a different type of architecture. As a result, we have gained some insight into using such an architecture for the execution of tasks in complex and dynamic domains: the most significant insight being related to real–time processing in the architecture.

In order to be able to specify a response time for a particular stimulus, each component of the architecture must perform within some temporal boundary. However, there are three areas where this is presently a problem: in processing by the capabilities, during the selection of a Domain Data Object, and when checking the triggering conditions of the REA's intentions. This is not to say that the REA architecture could not be used to do real–time execution. The point of this section is to identify why at present all responses are not guaranteed to be given in bounded time.

There are two problems with the processing capabilities. First, they are not interruptible. Once a capability (i.e., knowledge source) begins processing it must complete before it relinquishes control[4]. In a real–time environment the capabilities would have to be interruptible. Second, the processing capabilities complete in arbitrary times. The difficulty in determining how long a particular capability will take to complete is dependent upon how much search it has to perform. The Synthesize capability is probably the most representative of this problem, so consider it for example. The time it takes to complete depends upon (1) how many tasks are in the task network and how many causal structure records are specified in the IACL synthesize message, and (2) how many different Domain Data Objects there are which represent a particular

[4] It is intended that in the general O–Plan architecture that processing within a knowledge source will be broken down into stages in order to allow a knowledge source to be interrupted at a stage boundary. However, interrupting the processing when not at a stage boundary poses the same problem.

task. That is, the TBL was designed to allow the REA to trade–off ways to carry out particular tasks based upon resources used, execution time, number of conditions, number of effects, and execution cost. With this information the REA would be more intelligent when selecting a way to achieve a particular task; however, this flexibility is not conducive to real–time processing.

We just discussed the problem with selecting a DDO to specify the behavior of a particular task for the Synthesize capability. However, this is not only a problem for that capability, but any time a DDO must be selected.

The problem with checking the triggering conditions for the intentions of the REA is the way in which they are checked. That is, each time new information is assimilated into the REA's World Model, the triggers of all intentions on the untriggered agenda are tested. This testing in arbitrary in two ways. First, the number of intentions on the untriggered agenda varies, so the number which must be tested varies. Second, the time to test a particular trigger condition varies as to the number of conditions which must be verified in the World Model.

9.4 Contributions

The focus of this research has been to develop a set of characteristics which describe rational behavior for complex and dynamic environments that will allow for the design of quantitatively better execution systems. This dissertation contributes towards that goal in the following ways.

Characterization of Rational Behavior. The first contribution is an explicit characterization of rational behavior. This characterization is a specification of the type of behavior that an agent architecture should provide if it is to yield an agent that behaves rationally in complex and dynamic environments over a wide variety of domains. We originally identified eight characteristics and validated that set by comparatively evaluating a representative sample of existing execution systems against that set. It included characteristics for guaranteed response, failure recovery, innate behavior, asynchronous events, weighing alternatives, change of focus, predictability, and temporal reasoning. We then extended that set to include the characteristics of adaptability and contin-

uous operation, and replaced the failure recovery characteristic with that of failure management to establish a more pragmatic definition. The final version of the characterization specifies ten characteristics that form the basis of a guideline for the design of subsequent execution systems.

Inter–Agent Communication Language. The second contribution is a communication language that provides information to an execution system and allow for the dynamic adaptation of the behavior of an execution system. The simple message types of IACL provide a significant contribution in demonstrating how an execution system can be tasked, acquire and assimilate new or modified domain knowledge, and dynamically adapt its processing knowledge and capabilities at run–time. The definition of the language also provides for the dynamic specification of new message types that allows the language to be extended or adapted for novel domains.

Failure Management. The third contribution is a method of synthesizing protection monitors from causal structure information and a means to allow these monitors to be used to identify potential execution failures. This latter concept allows the execution system to detect potential failures early so as to provide a planning system (which must address the failure) with a greater amount of time to initiate a repair plan. Additionally, the concept of active sensing was introduced to show how this technique can be guaranteed to identify potential execution failures within specific temporal horizons.

Flexible Architecture. The fourth contribution is the validation of an agenda–based architecture as a flexible means to integrate characteristics of rationality (i.e., the O–Plan architecture). The architecture is dynamically adaptable and modular. This architecture uses knowledge sources to represent processing capabilities, and provides underlying mechanisms for asynchronous control necessary for reactive execution. The architecture is independent of the representation and thus, can be used to explore a variety of control strategies.

The Pacifica Simulator. The fifth and final contribution is a complex and dynamic environment for testing agent designs. It provides an environment for studying execution systems in the context of transportation logistics problems, and allows for remote sensing, complex object interactions, continuous time, dynamic reporting of task com-

pletion and failure, asynchronous task execution, and probabilistic task durations that are of extended length. Along with the characterization, the testbed provided by the Pacifica Simulator will allow us to quantitatively evaluate execution systems.

9.5 Conclusions

This research began with the desire to address some of the open research issues related to execution system design. It soon became apparent however, that there was no single example of a system that could be used as *the* model for a design since different execution systems offered different advantages. Thus, the efforts transformed from designing a new system to that of contributing to the way in which such systems could be better designed. In examining the AI literature to learn how to design execution systems, patterns began to emerge. Similar characteristics were identified which existed, in various guises, across implementations. This fact lead to the development of the characterization of rational behavior and the design methodology presented in this research.

The value of this characterization comes from the fact, that for the first time, we have a basis upon which to comparatively evaluate one execution system against another. However, the characterization falls short in several areas. First, it does not allow us to determine which characteristics are more important than others. Second, it does not provide insight into the possible conflicts which might arise when combining these characteristics in a particular architecture. What it is hoped can be taken away from this research is that the characterization does indeed define the minimum set of behaviors necessary to behave rationally. Additionally, that the ways in which these characteristics were implemented in the REA design does demonstrate that the specific design approach is one worth duplicating. What we need now is a way to determine the best implementation of a particular characteristic so we can design better architectures for rational, competent behavior in dynamic environments.

But have we truly characterized rationality? Probably not, since it could not be said that a system was irrational if it did not possess all of the identified characteristics. Perhaps a better question would be to ask whether a system should or needs to behave

rationally in complex and dynamic environments to be successful in accomplishing its tasks. These questions are left open to discussion. What has been learned as a result of this research is that you must have failures in order to have progress. Maybe this attempt at quantifying rationality is incomplete, but if that allows someone else to ponder these issues and develop a better way to design and compare execution systems then this work been successful. It is clear that a great deal of work remains to be done.

Bibliography

[Agre & Chapman 87] P. Agre and D. Chapman. Pengi: An Implementation of a Theory of Activity. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pages 268–272, 1987.

[Allen 79] J. F. Allen. A Plan-Based Approach to Speech Act Recognition. Technical Report 131, Department of Computer Science, University of Toronto, 1979.

[Allen 81] J. F. Allen. An Interval-Based Representation of Temporal Knowledge. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 741–747, 1981.

[Alterman *et al.* 91] R. Alterman, T. Carpenter, and R. Zito-Wolf. An Architecture for Understanding in Planning, Action, and Learning. *SIGART Bulletin*, 2(4):14–19, 1991.

[Ambros-Ingerson & Steel 88] J.A. Ambros-Ingerson and S. Steel. Integrating Planning, Execution and Monitoring. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 83–88, 1988.

[Appelt 81] D. Appelt. *Planning Natural Language Utterances to Satisfy Multiple Goals.* Unpublished PhD thesis, Stanford University, 1981.

[Art94] Artificial Intelligence Applications Institute, University of Edinburgh. *O-Plan Task Formalism Manual (Version 2.2)*, July 1994.

[Boddy & Dean 89] M. Boddy and T. Dean. Solving Time-dependent Planning Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 979–984, 1989.

[Boddy 91] M. Boddy. Temporal Reasoning for Planning and Scheduling. *SIGART Bulletin*, 4(3), 1991.

[Bonasso & Barratt 93] R. P. Bonasso and J. Barratt. A Reactive Robot System for Find and Visit Tasks in a Dynamic Ocean Environment. In *Proceedings of the 8th International Symposium on Unmanned Untethered Submersible Technology*, September 1993. IEEE Oceanographic Society.

[Bonasso & Kortenkamp 94] R. P. Bonasso and D. Kortenkamp. An Intelligent Agent Architecture In Which to Pursue Robot Learning. In *Proceedings of MCL-COLT '94 Robot Learning Workshop*, July 1994.

[Bonasso & Slack 92] R. P. Bonasso and M. Slack. Ideas on a System Design for End-User Robots. In *Proceedings of SPIE's Cooperative Intelligent Robotics in Space III at OE/Technology*, Boston, 1992.

[Bresina & Drummond 90] J. Bresina and M. Drummond. Integrating Planning and Reaction: A Preliminary Report. In *Proceedings of AAAI Spring Symposium (session on Planning in Uncertain, Unpredictable, or Changing Environments)*, 1990.

[Bresina *et al.* 93] J. Bresina, M. Drummond, and S. Kedar. *untitled*, chapter 6: Reactive, Integrated Systems Pose New Problems for Machine Learning. Morgan-Kaufmann, 1993.

[Brooks 86] R. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[Burnell 94] L. J. Burnell. Decision-Theoretic Control of Failure Recovery. In Kristian Hammond, editor, *Proceedings of the Conference on Artificial Intelligence Planning Systems*, pages 146–151, Chicago, IL, 1994.

[Chapman & Agre 86] D. Chapman and P. Agre. Abstract Reasoning as Emergent from Concrete Activity. In *Proceedings of Workshop on Planning and Reasoning about Action*, Portland, Oregon, 1986.

[Chapman 87] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32:333–377, 1987.

[Charniak *et al.* 87] E. Charniak, C. K. Riesbeck, D. V. McDermott, and J. R. Meehan. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, New Jersey, 1987.

[Cohen & Levesque 90] P. Cohen and H. J. Levesque. Rational Interaction as the Basis for Communication. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication.* MIT Press, 1990.

[Cohen 91] P. Cohen. A Survey of the Eighth National Conference on Artificial Intelligence: Pulling Together or Pulling Apart? *AI Magazine*, 12(1), 1991.

[Collins *et al.* 91] G. Collins, L. Birnbaum, B. Krulwich, and M. Freed. Model-Based Integration of Planning and Learning. *SIGART Bulletin*, 2(4):56–60, 1991.

[Connell 92] J. H. Connell. SSS: A Hybrid Architecture Applied to Robot Navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1992.

[Currie & Tate 91] K.W. Currie and A. Tate. O-Plan: the Open Planning Architecture. *Artificial Intelligence*, 51(1), 1991.

[Dean 84] T. Dean. Planning and Temporal Reasoning under Uncertainty. In *Proceedings of the IEEE Workshop on Principles of Knowledge Based Systems*, pages 210–213, 1984.

[Dimbleby & Burton 92] R. Dimbleby and G. Burton. *More than Words: An Introduction to Communication.* Routledge, London, second edition, 1992.

[Downs & Reichgelt 92] J. Downs and H. Reichgelt. CARP: An agent architecture integrating Classical And Reactive Planners. In *Proceedings of Eleventh Workshop of the UK Planning Special Interest Group*, Univ. of Sussex, Brighton, April 1992.

[Doyle *et al.* 86] R. Doyle, D. Atkinson, and R. Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-86)*, Philadelphia, PA, 1986. Morgan Kaufmann.

[Drabble *et al.* 94] B. Drabble, A. Tate, J. Dalton, and G. A. Reece. Conditions Types in O-Plan2: A Discussion. Technical Report ARPA-RL/O-Plan2/DN/1, Artificial Intelligence Applications Institute, 1994.

[Drummond & Bresina 90a] M. Drummond and J. Bresina. Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*, Boston, MA, August 1990. Morgan Kaufmann.

[Drummond & Bresina 90b] M. Drummond and J. Bresina. Planning for Control. Technical Report FIA-91-18, NASA Ames Research Center, 1990.

[Drummond & Kaelbling 90] M. E. Drummond and L. P. Kaelbling. Integrated Agent Architectures: Benchmark Tasks and Evaluation Metrics. In Katia P. Sycara, editor, *Proceedings of Innovative Approaches to Planning, Scheduling and Control*, pages 408–411, San Diego, California, November 5-8 1990.

[Drummond & Levinson 92] M. E. Drummond and R. Levinson. How Planning Can Help a Reactive System. Draft NASA Technical report as of January 1992, 1992.

[Drummond 89] M. Drummond. Situated Control Rules. In *Proceedings of Rochester Planning Workshop*, pages 18–33, October 1989.

[Durfee *et al.* 94] E. H. Durfee, P. Gmytrasiewicz, and J. S. Rosenschein. The Utility of Embeded Communications and the Emergence of Protocols. In *Proceedings of the AAAI-94 Workshop on Planning for Inter-agent Communication*, Seatle, WA, 1994.

[Elsaesser & Slack 94] C. Elsaesser and M. G. Slack. Deliberative Planning in a Robot Architecture. In *Proceedings of the AAIA/NASA Conference on Intelligent Robotics in Field, Factory, Service, and Space*, March 1994.

[Engelson & Bertani 92] S. P. Engelson and N. Bertani. ARS MAGNA: The Abstract Robot Simulator Manual. Technical report, Dept. of Computer Science, Yale University, 1992.

[Fikes 71] R.E. Fikes. Monitored Execution of Robot Plans Produced by STRIPS. In *Proceedings of IFIP Congress 71*, Ljubljana, Yugoslavia, August 23-28 1971. also SRI-TN-55.

[Fikes *et al.* 72a] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Learning and Executing Generalized Robot Plans. *Artificial Intelligence*, 3:251–288, 1972.

[Fikes *et al.* 72b] R.E. Fikes, P.E. Hart, and N.J. Nilsson. Some New Directions in Robot Problem Solving. *Machine Intelligence*, 7, 1972.

[Finin *et al.* 92] T. Finin, J. Weber, G. Wiederhold, G. Genesereth, M. Fritzson, R. McKay, D. McGuire, J. Pelavin, P. Shapiro, and C. Beck. Specification of the KQML Agent-Communication Language. Technical Report EIT TR 92-04, Enterprise Integration Technologies, Palo Alto, CA, 1992.

[Firby & Hanks 87] R. J. Firby and S. Hanks. The Simulator Manual. Technical Report YALEU/CSD/RR/563, Computer Science Dept., Yale University, 1987.

[Firby 87] R. J. Firby. An Investigation into Reactive Planning in Complex Domains. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, pages 202–206. Morgan Kaufmann, 1987.

[Firby 89] R. J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* Unpublished PhD thesis, Yale University, 1989.

[Firby 92] R. James Firby. Building Symbolic Primitives with Continuous Control Routines. In James Hendler, editor, *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 62–69, College Park, Maryland, 1992.

[Gat 91] E. Gat. *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots.* Unpublished PhD thesis, Virginia Polytechnic Institue and State University, 1991.

[Gat 92] E. Gat. Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-92).* Morgan Kaufmann, 1992.

[Georgeff & Ingrand 89] M. Georgeff and F. Ingrand. Decision-making in an Embedded Reasoning System. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, 1989.

[Georgeff & Lansky 87a] M. Georgeff and A. Lansky. Procedural Knowledge. Technical Report TN-411, SRI International, 1987.

[Georgeff & Lansky 87b] M. Georgeff and A. Lansky. Reactive Reasoning and Planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-87).* Morgan Kaufmann, 1987.

[Georgeff 83] M. Georgeff. Communication and Interaction in Multi–Agent Planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-83)*, pages 125–129. Morgan Kaufmann, 1983.

[Georgeff 89] M. Georgeff. An Embeded Real-Time Reasoning System. In *Proceedings of Rochester Planning Workshop*, October 1989.

[Georgeff 90] M. Georgeff. Situated Reasoning and Rational Behaviour. In *Proceedings of First Pacific Rim International Conference on Artificial Intelligence (PRICAI-90)*, Nagoya, Japan, 1990. also AAII-TN-21.

[Ginsberg 89] M. Ginsberg. Universal Planning: A(n Almost) Universally Bad Idea. AI *Magazine*, 10(4):40–44, Winter 1989.

[Gmytrasiewicz *et al.* 91] P. Gmytrasiewicz, E. H. Durfee, and D. K. Wehe. The Utility of Communication in Coordinating Intelligent Agents. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-91)*, pages 166–172, 1991.

[Gruber 93] T. R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. Technical Report KSL-93-04, Stanford University, 1993. Presented at the International Workshop on Formal Ontology, March, 1993, Padova, Italy.

[Hallam 94] J. Hallam, August 1994. Private Communication.

[Hanks & Firby 90] S. Hanks and R. J. Firby. Issues and Architectures for Planning and Execution. In Katia P. Sycara, editor, *Proceedings of Innovative Approaches to Planning, Scheduling and Control*, pages 59–70, San Diego, California, November 5-8 1990.

[Hanks 90] S. Hanks. Practical Temporal Projection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*, pages 158–163, Boston, Mass., 1990.

[Hanks 92] S. Hanks. Modeling a Dynamic and Uncertain World I: Symbolic and Probabilistic Reasoning about Change. *Artificial Intelligence*, 66(1):1–55, 1992.

[Hanks *et al.* 93] S. Hanks, M. E. Pollack, and P. R. Cohen. Benchmarks, Test Beds, Controlled Exerimentation, and the Design of Agent Architectures. *AI Magazine*, 14(4):17–42, 1993.

[Hart *et al.* 90] D. Hart, S. Anderson, and P. Cohen. Envelopes as a Vehicle for Improving the Efficiency of Plan Execution. In *Proceedings of DARPA Workshop on Innovative Approaches to Planning Scheduling and Control*, pages 71–76, San Diego, California, 1990. Morgan Kaufmann.

[Hendler 90] J. Hendler. Abstraction and Reaction. In James Hendler, editor, *Proceedings of the AAAI Spring Workshop on Planning in Uncertain, Unpredictable or Changing Environments*, University of Maryland, 1990.

[Hon92] Honeywell Systems & Research Center, 3660 Technology Dr., Minneapolis, MN 55418. *β-TMM Manual*, 1992. Honeywell SRC Technical Report CS-R92-012.

[Horvitz *et al.* 89] E. Horvitz, G. Cooper, and D. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 1121–1127, 1989.

[Howe & Cohen 91] A. Howe and P. Cohen. Failure Recovery: A Model and Experiments. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-91)*, pages 801–808. AAAI Press, 1991.

[ISX93] ISX Corporation. *ARPI KRSL Reference Manual 2.0.2*, February 1993. Nancy Lehrer (ed.).

[Kaelbling 88] L.P. Kaelbling. Goals as Parallel Program Specifications. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 60–65, St. Paul, Minnesota, 1988.

[Kaelbling 90] L. P. Kaelbling. Specifying Complex Behavior for Computer Agents. In Katia P. Sycara, editor, *Proceedings of Innovative Approaches to Planning, Scheduling and Control*, pages 433–438, San Diego, California, November 5-8 1990.

[Kaelbling 91] L. P. Kaelbling. An Adaptable Mobile Robot. In *Proceedings of the First European Conference on Artificial Life*, pages 41–47, Paris, France, December 1991.

[Kambhampati 90] S. Kambhampati. A Theory of Plan Modification. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*, pages 176–182, Boston, MA., July 29 – August 3 1990. Morgan Kaufmann.

[Kohout 93] R. Kohout. Representing Reactive Competences for use in Hard Real–Time Systems (Ph.D. Dissertation Proposal). Technical report, University of Maryland, 1993.

[Konolige & Nilsson 80] K. Konolige and N. Nilsson. Multiple–Agent Planning Systems. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-80)*, pages 138–141, Stanford, CA, 1980. Morgan Kaufmann.

[Laffey *et al.* 88] T. Laffey, P. Cox, J. Schmidt, S. Kao, and J. Read. Real-time Knowledge-based Systems. *AI Magazine*, 9(1):27–45, 1988.

[Lewis 91] L. Lewis. A Time-Ordered Architecute for Integrating Reflexive and Deliberative Behavior. *SIGART Bulletin*, 2(4):110–114, 1991.

[Lochbaum 94] K. E. Lochbaum. A Model of Plans to Support Inter-agent Communication. In *Proceedings of the AAAI-94 Workshop on Planning for Inter-agent Communication*, Seatle, WA, 1994.

[Lyons & Arbib 89] D. M. Lyons and M. A. Arbib. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Trans. on Robotics and Automation*, 5(3):280–293, 1989.

[Lyons & Hendriks 92] D. M. Lyons and A. J. Hendriks. A Practical Approach to Integrating Reaction and Deliberation. In *Proceedings of First International Conference on Artificial Intelligence Planning Systems*, 1992.

[Lyons 91] D. M. Lyons. Representing and Analysing Plans as Networks of Concurrent Processes. *IEEE Trans. on Robotics and Automation*, 1991.

[Lyons *et al.* 91] D. M. Lyons, A. J. Hendriks, and S. Mehta. Achieving Robustness by Casting Planning as Adaptation of a Reactive System. In *Proceedings of IEEE International Conference on Robotics & Automation*, 1991.

[Martin & Firby 91] C. Martin and R. J. Firby. An Integrated Architecture for Planning and Learning. *SIGART Bulletin*, 2(4):125–129, 1991.

[McAllester & Rosenblitt 91] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-91)*, 1991.

[McDermott 76] D. McDermott. *Flexibility and Efficiency in a Computer Program for Designing Circuits.* Unpublished PhD thesis, Dept. of Electrical Engineering, MIT, 1976.

[McDermott 92] D. McDermott. Transformational Planning of Reactive Behavior. Technical Report YALEU/CSD/RR941, Yale University, 1992.

[Miller 85] D. P. Miller. Planning by Search Through Simulations. Technical Report YALEU/CSD/RR423, Yale University, 1985.

[Minton 91] S. Minton. On Modularity in Intelligent Integrated Architectures. *SIGART Bulletin*, 2(4):134–135, 1991.

[Musliner *et al.* 91] D. J. Musliner, E. H. Durfee, and K. G. Shin. Execution Monitoring and Recovery Planning with Time. In *Proceedings of the Conference on Artificial Intelligence Applications*, 1991.

[Musliner *et al.* 93] D. J. Musliner, E. H. Durfee, and K. G. Shin. A Cooperative Intelligent Real–Time Control Architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1993.

[Nguyen *et al.* 93] D. Nguyen, S. Hanks, and C. Thomas. The TRUCKWORLD Manual. Technical Report 93-09-08, Dept. of Computer Science & Engineering, University of Washington, 1993.

[Norvig 92] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, San Mateo, CA, 1992.

[Perrault 90] C. R. Perrault. An Application of Default Logic to Speech Act Theory. In P. R. Cohen, J. Morgan, and M. E. Pollack, editors, *Intentions in Communication.* MIT Press, 1990.

[Pollack & Ringuette 90] M. E. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-90)*, pages 193–189, 1990.

[Pollack 86] M. E. Pollack. *Inferring Domain Plans in Question Answering.* Unpublished PhD thesis, Department of Computer Science, University of Pennsylvania, 1986.

[Pryor 94] L. M. Pryor. *Opportunities and Planning in an Unpredictable World.* Unpublished PhD thesis, Northwestern University, 1994. Also available as TR-53 from The Institute for the Learning Sciences.

[Reece & Tate 93] G. A. Reece and A. Tate. The Pacifica NEO Scenario. Technical Report ARPA-RL/O-Plan2/TR/3, Artificial Intelligence Applications Institute, March 1993.

[Reece *et al.* 93] G. Reece, A. Tate, D. Brown, and M. Hoffman. The PRECiS Environment. In *Proceedings of National Conference on Artificial Intelligence (AAAI-93) DARPA-RL Planning Inititive Workshop*, Washington, D.C., 1993. Available as ARPA-RL/O-Plan2/TR/11 from the Artifical Intelligence Applications Institute.

[Rosenschein & Kaelbling 86] S.J. Rosenschein and L.P. Kaelbling. The Synthesis of Digital Machines with Provable Epistemic Properties. In *Proceedings of Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–98, Asilomar, California, 1986. also SRI-TN-412.

[Rosenschein 87] S.J. Rosenschein. Formal Theories of Knowledge in AI and Robotics. Technical report, Center for Study of Language and Information, Stanford, California, 1987.

[Sacerdoti 77] E.D. Sacerdoti. *A Structure for Plans and Behavior.* Elsevier-North Holland, New York, 1977.

[Sanborn & Hendler 88] J.C. Sanborn and J.A. Hendler. A Model of Reaction for Planning in Dynamic Environments. *Artificial Intelligence in Engineering*, 3(2):95–102, 1988.

[Schoppers & Shu 90] M. Schoppers and R. Shu. An Implementation of Indexical/Functional Reference for Embedded Execution of Symbolic Plans. In Katia P. Sycara, editor, *Proceedings of DARPA Workshop on Innovative Approaches to Planning Scheduling and Control*, San Diego, California, 1990. Morgan Kaufmann.

[Simmons 90] R. G. Simmons. An Architecture for Coordinating Planning, Sensing, and Action. In *Proceedings of DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 292–297, San Diego, CA, 1990.

[Simmons 92] R. G. Simmons. Monitoring and Error Recovery for Autonomous Walking. In *Proceedings of IEEE International Workshop on Intelligent Robots and Systems*, pages 1407–1412, Raleigh, NC, 1992.

[Simmons 93] R. G. Simmons. Structured Control for Autonomous Robots. *IEEE Transactions on Robotics and Automation*, 1993.

[Simmons 94] R. G. Simmons, July 1994. Private E-Mail Communication.

[Slack 92] M. Slack. Sequencing Formally Defined Reactions for Robotic Activity: Integrating RAPS and GAPPS. In *Proceedings of SPIE's Sensor Fusion at OE/Technology*, Boston, 1992.

[Sussman 75] G. Sussman. *A Model of Skill Acquisition.* Elsevier Scientific, 1975.

[Suthers 94] D. D. Suthers. Planning for Interagent Communication: What's Special? In *Proceedings of the AAAI-94 Workshop on Planning for Inter-agent Communication*, Seatle, WA, 1994.

[Tate 77] A. Tate. Generating Project Networks. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-77)*, 1977.

[Tate 84] A. Tate. Planning and Condition Monitoring in a FMS. In *Proceedings of International Conference on Flexible Automation Systems*, London, England, 1984. Available

as AIAI-TR-2 from the Artifical Intelligence Applications Institute.

[Tate 89] A. Tate. Coordinating the Activities of a Planner and an Execution Agent. In G. Rodriguez, editor, *Proceedings of NASA Conference on Space Telerobotics*, Jet Propulsion Laboratory, California, 1989. JPL Publications. also AIAI-TR-57.

[Tate 93] A. Tate. Authority Management Coordination between Task Assignment Planning and Execution. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-93) Workshop on Knowlege-based Production Planning, Scheduling and Control*, Chambery, France, 1993. Available as AIAI-TR-133 from the Artifical Intelligence Applications Institute.

[Tate 94] A. Tate. The Emergence of "Standard" Planning and Scheduling System Components—Open Planning and Scheduling Architectures. In C. Bäckström and E. Sandwell, editors, *Current Trends in AI Planning*, pages 14–32. IOS Press, 1994. Proceedings of the European Workshop on Planning.

[Tate *et al.* 92] A. Tate, B. Drabble, and R. Kirby. Spacecraft Command and Control Using AI Planning Techiques – The O-Plan2 Project – Final Report. Technical report, USAF Rome Laboratory RL-TR-92-217, 1992. Available as AIAI-TR-109 from Artifical Intelligence Applications Institute.

[Tate *et al.* 94] A. Tate, B. Drabble, and R. Kirby. O-Plan2: an Open Architecture for Command, Planning and Control. In M. Fox and M. Zweben, editors, *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[Turner 94] E. H. Turner. Selecting Information to Communicate. In *Proceedings of the AAAI-94 Workshop on Planning for Inter-agent Communication*, Seatle, WA, 1994.

[Vere & Bickmore 90] S. Vere and T. Bickmore. The Basic Agent. *Computational Intelligence*, 6(1), 1990.

[Whitehead 91] S. Whitehead. A framework for integrating perception, action, and trial-and-error learning. *SIGART Bulletin*, 2(4):174–178, 1991.

[Wilkins & Myers 94] D. E. Wilkins and K. L. Myers. A Common Knowledge Representation for Plan Generation and Reactive Execution. *Logic and Computation*, 1994. Accepted, to appear late 1994, 1995.

[Wilkins 84] D.E. Wilkins. Domain-Independent Planning: Representation and Plan Generation. *Artificial Intelligence*, 22(3):269–301, 1984. Available as SRI-TR (May 1983) from SRI International.

[Wilkins 85a] D.E. Wilkins. *Practical Planning.* Morgan Kaufmann, 1985.

[Wilkins 85b] D.E. Wilkins. Recovering from Execution Errors in SIPE. *Computational Intelligence*, 1:33–45, 1985.

[Wilkins 93a] D. E. Wilkins. Planning in Uncertain and Dynamic Environments. From DARPA Planning Initiative Workshop, San Antonio, Texas, February 1993.

[Wilkins 93b] D. E. Wilkins. Representing Knowledge and Plans as ACTs. working document, February 1993.

Appendix A

IACL Synthesize Message

This appendix contains the IACL synthesize message for the Small Scale NEO.

```
(:synthesize-new
 (node-network
  ((NODE-3 (FLY-TRANSPORT C5-1 Delta City-K) ())
   (NODE-4-1 (DRIVE GT2 Abyss Delta) ())
   (NODE-4-2 (LOAD GT2 Abyss PASSENGERS) ())
   (NODE-4-3 (DRIVE GT2 Delta Abyss) (DRIVE-1))
   (NODE-4-4 (UNLOAD GT2 Delta) ())
   (NODE-5-1 (DRIVE GT2 Barnacle Delta) (DRIVE))
   (NODE-5-2 (LOAD GT2 Barnacle PASSENGERS) ())
   (NODE-5-3 (DRIVE GT2 Delta Barnacle) (DRIVE))
   (NODE-5-4 (UNLOAD GT2 Delta) ())
   (NODE-6-1 (DRIVE GT1 Calypso Delta) ())
   (NODE-6-2 (LOAD GT1 Calypso PASSENGERS) ())
   (NODE-6-3 (DRIVE GT1 Delta Calypso) (DRIVE))
   (NODE-6-4 (UNLOAD GT1 Delta) ())
   (NODE-7 (FLY-PASSENGERS B707 City-K Delta) ())
   (NODE-8 (FLY-TRANSPORT C5-1 City-K Delta) ())))
 (priority 1)
 (orderings
  ((NODE-3 nil (NODE-5-1 NODE-6-1))
   (NODE-4-1 (NODE-5-4) (NODE-4-2))
   (NODE-4-2 (NODE-4-1) (NODE-4-3))
   (NODE-4-3 (NODE-4-2) (NODE-4-4))
   (NODE-4-4 (NODE-4-3) (NODE-7))
   (NODE-5-1 (NODE-3) (NODE-5-2))
   (NODE-5-2 (NODE-5-1) (NODE-5-3))
   (NODE-5-3 (NODE-5-2) (NODE-5-4))
   (NODE-5-4 (NODE-5-3) (NODE-4-1))
   (NODE-6-1 (NODE-3) (NODE-6-2))
   (NODE-6-2 (NODE-6-1) (NODE-6-3))
   (NODE-6-3 (NODE-6-2) (NODE-6-4))
```

```
  (NODE-6-4 (NODE-6-3) (NODE-7))
  (NODE-7 (NODE-4-4 NODE-6-4) (NODE-8))
  (NODE-8 (NODE-4-4 NODE-6-4) nil)))
(gost
 ((CSTR-1  (AT GT1) DELTA (NODE-6-1) (NODE-3) :FLEX)
  (CSTR-2  (AT GT2) DELTA (NODE-5-1) (NODE-3) :FLEX)
  (CSTR-3  (RES-STATUS GT1) AVAILABLE (NODE-6-1) (NODE-3) :FLEX)
  (CSTR-4  (RES-STATUS GT2) AVAILABLE (NODE-5-1) (NODE-3) :FLEX)
  (CSTR-5  (AT C5-1) DELTA (NODE-8) (NODE-3) :FLEX)
  (CSTR-6  (AT GT2) ABYSS (NODE-4-2) (NODE-4-1) :FLEX)
  (CSTR-7  (AT GT2) DELTA (NODE-4-4) (NODE-4-3) :FLEX)
  (CSTR-8  (AT GT2) DELTA (NODE-8) (NODE-4-3) :FLEX)
  (CSTR-9  (AT GT2) DELTA (NODE-8) (NODE-4-4) :FLEX)
  (CSTR-10 (AT GT2) BARNACLE (NODE-5-2) (NODE-5-1) :FLEX)
  (CSTR-11 (AT GT2) DELTA (NODE-4-1) (NODE-5-3) :FLEX)
  (CSTR-12 (AT GT2) DELTA (NODE-5-4) (NODE-5-3) :FLEX)
  (CSTR-13 (RES-STATUS GT2) AVAILABLE (NODE-4-1) (NODE-5-4) :FLEX)
  (CSTR-14 (AT GT1) CALYPSO (NODE-6-2) (NODE-6-1) :FLEX)
  (CSTR-15 (AT GT1) DELTA (NODE-6-4) (NODE-6-3) :FLEX)
  (CSTR-16 (AT GT1) DELTA (NODE-8) (NODE-6-3) :FLEX)
  (CSTR-17 (AT GT1) DELTA (NODE-8) (NODE-6-4) :FLEX))))
```

Appendix B

The Supervise Capability Algorithm

This appendix contains the processing algorithm of the Supervise capability.

The Supervise capability accepts Task Directive Objects with an execution status of ready (r), processing (p), or failed (f)...

- With e-status = "r"
  1. Find all of the high-level tasks that are eligible to execute by the fact that their ordering constraints are satisfied.
  2. Of those tasks that are eligible to be executed, determine which are cleared for execution by the fact that their pre-conditions are satisfied by querying the World Model.
  3. For each high-level task cleared for execution a Procedure needs to be selected that specifies the how the task will be carried out in the environment.
  4. Each selected Procedure is then posted to the AM as an AE to be processed by the Execute capability.
  5. The Task Directive Object and thus, the remaining high-level tasks, are suspended by posting the Task Directive Object to the AM with a trigger that will prevent it from becoming an active intention until the effects all of the Procedures posted in (4.) are satisfied according to the World Model. The e-status of the Task Directive Object is changed to "p."
- With e-status = "p"
  1. Check the expected effects of the high-level tasks that were executing to determine which tasks have successfully completed. This information is then stored in the appropriate Task Directive Object to the tasks.
  2. The e-status of the Task Directive Object is changed to "r" and it is posted to the AM for processing by the Supervise capability.
- With e-status = "f"
  1. Halt all processing of the Task Directive Object.

Appendix C

The Execute Capability Algorithm

C.1 Execute Capability Algorithm (Part I)

The Execute capability accepts primitive and network Procedure Objects with an execution status of ready (r), or processing (p)...

- With e-status = "r"
  - Primitive Procedure
    1. Determine if the effects of the Procedure are already satisfied by querying the World Model.
    2. If the effects are not satisfied, then change the e-status to "p" and post the Procedure to the AM for processing by the Dispatch capability.
    3. If the effects are satisfied, then do not execute the Procedure.
  - Network Procedure
    1. For each subtask of the network:
       (a) Find all Domain Data Objects that match the pattern of the task.
       (b) Select a DDO and verify that its conditions are satisfied according to the World Model.
       (c) Synthesize Procedure Objects for the procedures of the selected DDO.
    2. Determine which subtasks of the network are eligible to execute according to their ordering constraints.
    3. Of those which are eligible, remove from consideration those that are temporally ordered after other subtasks.
    4. Time stamp the beginning of execution for the network.
    5. For each eligible subtask:
       (a) Select a procedure
       (b) Set the procedure's e-status to "r"

(c) Post it to the AM for processing by the Execute capability.

6. Suspend execution of the remaining subtasks in the network by posting the Procedure Object to the AM with a trigger that will prevent it from becoming an active intention until the effects of all of the subtasks posted in (5.) are satisfied according to the World Model. The e-status of the Procedure Object is changed to "p."

C.2 Execute Capability Algorithm (Part II)

The Execute capability accepts primitive and network Procedure Objects with an execution status of ready (r), or processing (p)...

- With e-status = "p"
  - Primitive Procedure
    1. Determine if the effects of the task have been realized by querying the World Model.
    2. If the end conditions of the task are satisfied then execution of the task is complete. Otherwise, determine if the task is one which can be repeated.
    3. If repeatable, then repeat the execution of the task. Otherwise, we have a failure.
  - Network Procedure
    1. For each subtask of the network that was dispatched:
       (a) Determine if the effects of the subtask have been realized by querying the World Model.
       (b) If the end conditions of the subtask are satisfied then time stamp the subtask so the Procedure can use this information for temporal reasoning.
       (c) Remove the subtask from the network and note which subtasks of the network have been successfully executed.
    2. If the network is now empty then make sure that the execution of the entire network has successfully achieved its effects by querying the World Model. If satisfied, then execution of the Network Procedure is complete. If not satisfied:
       (a) Determine if the task represented by the network is repeatable.
       (b) If so, then repeat its execution.
       (c) Otherwise, we have a failure.

C.3 Execute Capability Algorithm (Part III)

The Execute capability accepts primitive and network Procedure Objects with an execution status of ready (r), or processing (p)...

- With e-status = "p"
  - Network Procedure
    1. If the network is not empty and all of the previously dispatched subtasks have successfully executed then:
       (a) For each task remaining in the Network Procedure determine if it is eligible to execute by the fact that its ordering constraints have been satisfied.
       (b) Check the temporal constraints to determine if any temporally constrained subtasks are now eligible for execution. If so, then add those subtasks to the eligible set.
       (c) If the eligible set is empty then we have a failure.
       (d) Otherwise, each subtask in the eligible set is posted to the AM for processing by the Execute capability and its e-status is changed to "r"
       (e) Suspend execution of the remaining subtasks in the network by posting the Procedure Object to the AM with a trigger that will prevent it from becoming an active intention until the effects of all of the subtasks posted in (d) are satisfied according to the World Model.
    2. If the network is not empty and all of the previously dispatched subtasks have not successfully executed then find the first subtask (since there could be more than one which was unsuccessful) that failed and determine if it is repeatable.
    3. If repeatable, then repeat its execution.
    4. Otherwise, we have a failure.